P2PS-NS2 Integration

P2P Discovery and Communication Within NS2

Ian J. Taylor

September 1, 2004

Contents

Chapter 1

Introduction

This chapter gives a very brief background into the motivation behind this work and describes the core nature of the network that the NS2 simulations that will be performed will be run on. Here, a brief overview of the architecture employed for this integration at concept is provided along with a description of how the various parts tie in together.

1.1 SRSS Group Overview

The key focus for the Scalable, Robust Self-Organizing Sensor (SRSS) systems group in NRL is to investigate and model, using network simulation tools, lightweight network application discovery mechanisms suitable for application in mobile sensor systems. The SRSS systems in question are envisioned to leverage self-organizing computer communication networks based on Mobile Ad-hoc Networking (MANET) routing protocols which operate using wireless communication links and have no centralized administration or control.

Each node in a MANET network participates in the discovery of a route and therefore low-level routing protocols are paramount to the overall behaviour. However, it is anticipated that middleware network services beyond routing will be required to facilitate autonomous self-organization of sensors and their various related data collection, processing, and reporting functions.

The actual sensors are relatively simple devices that consist of a CPU, a data collection mechanism e.g. ADC convertor for audio, images etc and a wireless network card for communication across the MANET network to other participating nodes in the community.

The complexity of middleware approaches being considered and examined range from utilization of simple, organic network services which might be pro-

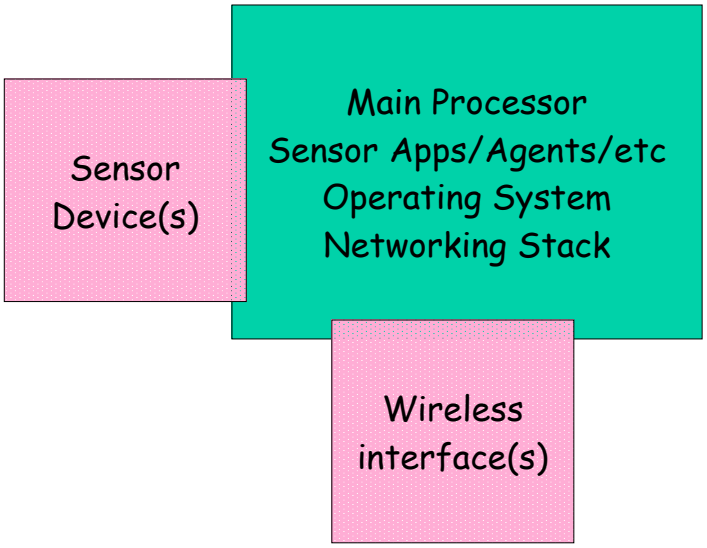

Figure 1.1: The components of a wireless sensor within an SRSS network.

vided by the network layer (network name/address resolution, IP multicast, ANYCAST) to potentially heavy-weight, highly stateful, complex agent-based architectures. The focus of this task will be lightweight (minimally complex) middleware discovery mechanisms and services which can facilitate publish and subscribe relationships among a set of sensor application peers participating in an SRSS network. The context of highly dynamic, possibly mobile, networking will place special challenges on such protocols ability to perform peer neighbor and service discovery and to maintain that information in the face of node outages and/or relocation within the network.

1.2 Creating P2P Discovery Simulations

It is paramount to the SRSS project that they construct a reusable architecture for testing out various discovery mechanisms employed by different middleware infrastructures. The architecture therefore contains an abstract P2P interface, which has been developed at Cardiff University within the Gridlab [5] and GridOneD projects [6], called the GAP. This interface is provides access to core P2P services, which were extracted from examining a number of P2P applications and extracting the functionality that most application need and was motivated from the GAT interface. These interfaces

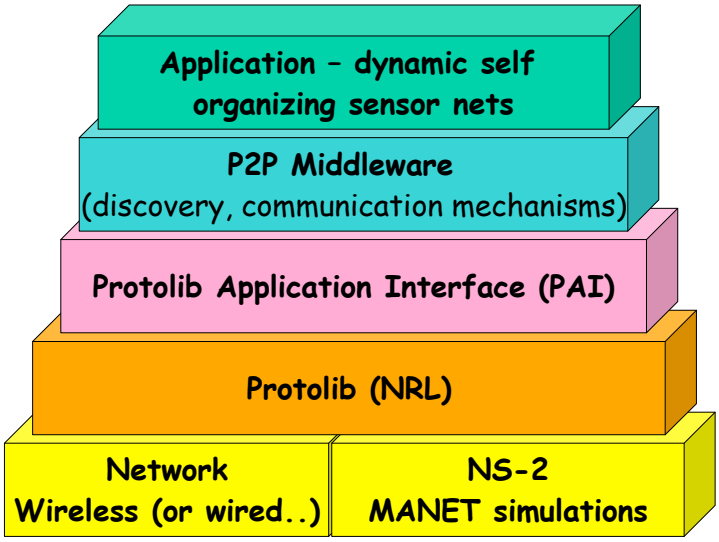

Figure 1.2: An overview of the P2PS-NS2 software stack

are described in the next two sections.

The resulting architecture is shown in Figure 1.2. The application (i.e. mobile sensors) should be able to ulitilise the discovery mechanisms from a P2P middleware layer, which interfaces through to Protolib via a generic interface, called PAI. This enables the resulting application to be deployed in a networked or an NS-2 simulation environment.

Protolib is a low-level communication, event dispatching and timing library that can be used on top of a network or within the NS-2 network simulator environment.

NS-2 [1] is a discrete event simulator that supports the link layer upwards on the OSI stack i.e. the network, transport, session, presentation and application layer, respectively. It can support both wired and wireless simulations and works on most platforms and therefore satisfies the main focus of the project, that is, to test out various P2P discovery and communication mechanisms within various network extremities.

1.2.1 The GAT

The GAT interface provides a generalised collection of calls to shield Grid applications from implementation details of the underlying Grid middleware, and is being developed in the European GridLab project [5]. The GAT

utilises *adaptors* that provide the specific bindings from the GAT interface to the underlying mechanisms that implement this functionality. For example, a *move_file* command may have many GAT adaptors that implement this functionality depending upon the particular execution environment used, such as GridFTP, JXTA pipes or a local *cp* command.

GAT may be referred to as *upperware*, which distinguishes it from middleware (which provides the actual implementation of the underlying functionality). Until recently, application developers typically interact with the middleware directly. However, it is becoming increasingly apparent that this transition from one type of middleware to another is not a trivial one. Using interfaces like GAT, migrating from one middleware environment to another is easier, and typically achieved by setting an environment variable. This is illustrated in the next section where we have implemented an adaptor to bind to P2P middleware for operating in P2P environments as well as the Grid environments supported directly by GridLab. This means that exactly the same Triana implementation can be used within both environments transparently.

1.2.2 The GAP Interface

The Grid Application Prototype Interface (GAP Interface) is a generic application interface providing a subset of the GAT functionality. It is middleware independent, with bindings provided for different Grid middleware such as JXTA and Web Services, as illustrated in Figure 1.3.

Part of the motivation behind the GAP Interface is as a stopgap to enable us to develop distribution mechanisms within Triana while the GridLab GAT is being developed. When the GridLab GAT becomes available the GAT-API will replace the GAP Interface within Triana and should enable Triana to make use of the advanced security, logging and other GridLab services. However, the GAP Interface will live on, both as a simple interface for prototyping Grid and P2P applications, and as an adaptor within the GridLab GAT architecture providing various discovery and communication capabilities. Currently there are three GAP bindings implemented:

JXTA - The original GAP Interface binding was to JXTA [4]. JXTA is a set of protocols for Peer-to-Peer discovery and communication originally developed by Sun Microsystems. Although we achieved some initial success with JXTA, we have since had problems with the speed and reliability of the JXTA binding.

P2PS - a lightweight Peer-to-Peer middleware. See Chapt. 7.

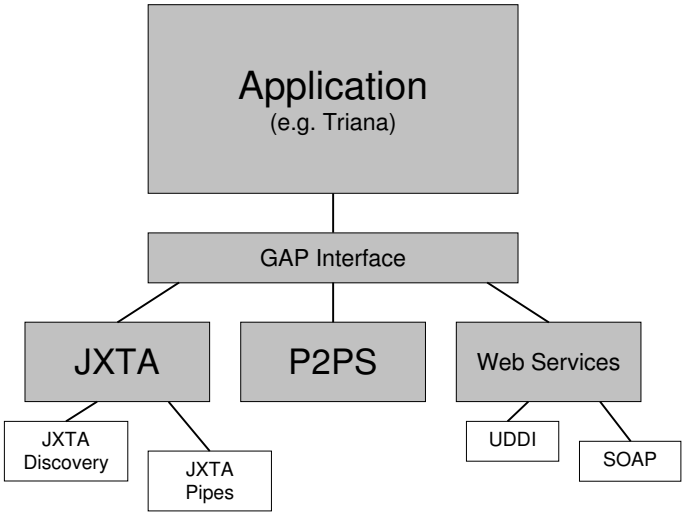

Figure 1.3: The GAP Interface provides a middleware independent interface for developing Grid applications

Web Services - The most recent GAP binding allows applications to discover and interact with Web Services – using the UDDI registry [12] and the Web Service Invocation Framework (WSIF) [13].

1.3 Pluggin in Java

The work described in this manual implements a Java framework for plugging in Java middleware, such as the GAP and bindings, such as Jxta and P2PS. The protoype implementation here interfaces directly to the P2PS middleware, which is described in detail in Chapt 7. This interfaces via the PAI interface (described in Chapt. 4) to use the Protolib application toolkit.

Since the middleware is written in Java, a JNI bridge is needed in order to map between the C++ NS2 objects and the associated Java objects. This bridging mechanism is required at both the input to the middleware and at its lower communication layers. The input to the middleware allows the C++ objects to access the Java functionality and is achieved by creating a Java Virtual Machine (JVM) within the C++ application. The lower communication levels of the middleware must then map to the PAI and Protolib C++ libraries so this communication layer must also be interface

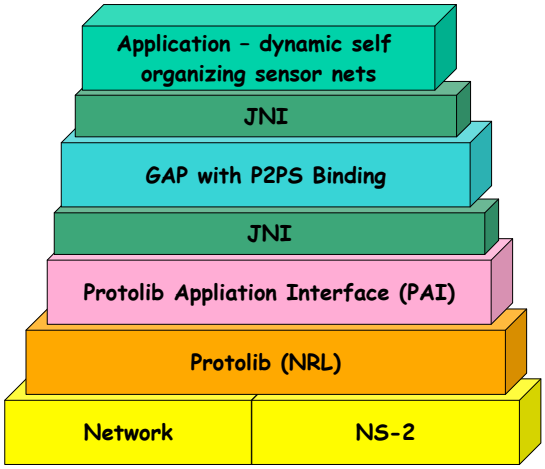

Figure 1.4: The SRTSS architecture showing the JNI interface to the Java objects that can be included within the NS2 simulations.

using JNI. The resulting arhictecture illustrating this is shown in Figure 1.4.

1.4 conclusion

In this chapter a brief motivation and background into the work described in this manual is given. The architecture is given and discussed in context to the Java middleware that is employed within the software stack utilised within this project.

Chapter 2

Installing the P2PS NS2 Toolkit

This chapter describes the installation of core packages needed in order to get the P2PS-NS2 toolkit operating. The core packages needed are:

1. **Protolib:** a core package for adding timers and UDP communication within NS2 [9]
2. **P2PS-NS2:** this includes a customized version of the P2PS middleware [8] and the PAI interface to Protolib, described in Chapt. 4 and the JNI interface for attaching Java Objects to NS2 nodes.

2.1 Installing the Protolib NS2 Binding

With the Protolib release there is a supplied Makefile for NS version 2.29 and a README.TXT file in the *ns* directory. The read me file describes the steps involved in installing protolob into NS2. They are as follows:

```
To use PROTOLIB with ns, you will need to at least
modify the ns "Makefile.in" to build the PROTOLIB
code into ns.  To do this, use the following steps:


1)  Make a link to the PROTOLIB source directory in the
    ns source directory.  (I use "protolib" for the link
    name in the steps below).

2)  Provide paths to the PROTOLIB include files by setting

    PROTOLIB_INCLUDES = -Iprotolib/common -Iprotolib/ns
```

```
    and adding $(PROTOLIB_INCLUDES) to the "INCLUDES" macro
    already defined in the  ns "Makefile.in"

3)  Define compile-time CFLAGS needed for the PROTOLIB code
    by setting

    PROTOLIB_FLAGS = -DUNIX -DNS2 -DPROTO_DEBUG -DHAVE_ASSERT

    and adding $(PROTOLIB_FLAGS) to the "CFLAGS" macro already
    defined in the ns "Makefile.in"

4)  Add the list of PROTOLIB object files to get compiled and linked
    during the ns build.  For example, set

    OBJ_PROTOLIB_CPP = \
            protolib/ns/nsProtoAgent.o protolib/common/protoSim.o\
            protolib/common/networkAddress.o \
            protolib/common/protocolTimer.o \
            protolib/common/debug.o

    and then add $(OBJ_PROTOLIB_CPP) to the list in the "OBJ"
    macro already defined in the ns "Makefile.in"

    Note: "nsProtoAgent.cpp" contains a starter ns agent which uses the
    PROTOLIB ProtocolTimer and UdpSocket classes.

5)  Add the the rule for .cpp files to ns-2 "Makefile.in":

    .cpp.o:
    @rm -f $@
    $(CC) -c $(CFLAGS) $(INCLUDES) -o $@ $*.cpp

    and add to the ns-2 Makefile.in "SRC" macro definition:

    $(OBJ_CPP:.o=.cpp)


6)  Run "./configure" in the ns source directory to create a new
    Makefile and  then type "make ns" to rebuild ns.


Brian Adamson
<mailto://adamson@itd.nrl.navy.mil>
18 December 2001
```

The first thing to take note is that Protolib is basically a plug-in for NS 2 to allowing a trigger mechanism, based on timers and a UDP socket implementation for passing data between NS2 nodes. Therefore, to install

this plug-in, you must recompile NS2. It is therefore advisable to build NS2 from scratch then add the protolib plug-in.

2.2 Installing the P2PS-NS2 Binding

The P2PS-NS2 toolkit follows a similar installation path to Protolib but instead of using a softlink, it uses environment variables within the *Makefile.in* file to point to the installation directory for the source code for P2PS-NS2. The installation is provided below and follows a similar style to the Protolib procedure for simplicity. This file can be found in the src/build/ns2PAIConfig directory within the P2PS-NS2 source tree.

```
To install P2PS-NS2 (and PAI) you need to install Protolib
and  modify the ns "Makefile.in" to build the P2PS-NS2
code into ns (there is a Makefile.in file for the ns2.26
release given in this directory). To do this, use the
following steps:

1)  Install Protolib

2)  Set a PAI environment variable to point to your installation
directory for P2PS-NS2 (built on top of PAI) and create
pointers to the various subdirectories for the source, as follows:

PAI = ../../../Apps/nrl/p2ps-ns2/src/pai

PAI_CORE = $(PAI)/core
PAI_FACTORY = $(PAI)/factory
PAI_IMPL = $(PAI)/impl
PAI_JNI = $(PAI)/jni

2)  Provide paths to the P2PS-NS2 include files by setting

PAI_INCLUDES = -I$(JAVA_HOME)/include -I$(PAI_CORE)
-I$(PAI_FACTORY) -I$(PAI_IMPL) -I$(PAI_IMPL)/ns -I$(PAI_JNI)

    and adding $(PAI_INCLUDES) to the "INCLUDES" macro
    already defined in the  ns "Makefile.in"

3)  Add the list of P2PS-NS2 object files to get compiled and linked
    during the ns build.  For example, set

OBJ_PAI_CPP = $(PAI_CORE)/LinkedList.o $(PAI_IMPL)/PAIDispatcher.o \
$(PAI_IMPL)/PAIEngine.o $(PAI_FACTORY)/PAIFactory.o \
$(PAI_CORE)/PAIMultipleListener.o $(PAI_IMPL)/PAISocket.o \
$(PAI_IMPL)/PAITimer.o $(PAI_IMPL)/ns/PAINS2UDPSocket.o \
```

```
$(PAI_IMPL)/ns/PAINS2Timer.o $(PAI_CORE)/PAIListener.o \
$(PAI_CORE)/PAI.o $(PAI_IMPL)/ns/PAIAgent.o\
$(PAI_IMPL)/ns/PAIBroker.o\
$(PAI_FACTORY)/PAIEnvironment.o\
$(PAI_JNI)/JVMRef.o $(PAI_IMPL)/ns/JavaAgent.o\
$(PAI_JNI)/JavaPAI.o $(PAI_JNI)/JavaEnv.o

    and then add $(OBJ_PAI_CPP) to the list in the "OBJ"
macro already defined in the ns "Makefile.in"

    Note: "JavaAgent.cpp" contains the ns agent for
integrating Java objects.

4)  Add the rule for .cpp files to ns-2 "Makefile.in":

    .cpp.o:
    @rm -f $@
    $(CC) -c $(CFLAGS) $(INCLUDES) -o $@ $*.cpp

    and add to the ns-2 Makefile.in "SRC" macro definition:

    $(OBJ_CPP:.o=.cpp)

    (note this has already been done - if you have installed
protolib correctly).

5)  Create a shared library - define compile-time SHARED
Library flags and  libraries needed for your platform to
create a shared library (this is needed for the JNI binding).
On my Mac OS 10.x, these are defined as follows:

PAI_LIB = -framework JavaVM
PAI_SHARED_LDFLAGS = $(SHARED)

    and adding $(PAI_LIB) to the "LIB" macro already
    defined in the ns "Makefile.in"

    and adding a new rule to make the shared library

libnspai.dyn: $(OBJ) common/tclAppInit.o
$(LINK) $(PAI_SHARED_LDFLAGS) -o $@ \
common/tclAppInit.o $(OBJ) $(LIB)


6)  Run "./configure" in the ns source directory to create a new
    Makefile

7) Type "make ns" to rebuild ns - this creates the static library
```

```
8) Type "make libnspai.dyn" to rebuild the dynamic library
needed for the installation of the JNI frameworks.
```

2.2.1 Building NS2

The resulting NS2 Makefile should therefore including both the Protolib and P2PS dependencies. A complete version of my Makefile, used to build NS 2 version 2.26 on an Apple Mac, is provided below:

```
#  Copyright (c) 1994, 1995, 1996
#  The Regents of the University of California.  All rights reserved.
#
#  Redistribution and use in source and binary forms, with or without
#  modification, are permitted provided that: (1) source code distributions
#  retain the above copyright notice and this paragraph in its entirety, (2)
#  distributions including binary code include the above copyright notice and
#  this paragraph in its entirety in the documentation or other materials
#  provided with the distribution, and (3) all advertising materials mentioning
#  features or use of this software display the following acknowledgement:
#  ‘‘This product includes software developed by the University of California,
#  Lawrence Berkeley Laboratory and its contributors.’’ Neither the name of
#  the University nor the names of its contributors may be used to endorse
#  or promote products derived from this software without specific prior
#  written permission.
#  THIS SOFTWARE IS PROVIDED ‘‘AS IS’’ AND WITHOUT ANY EXPRESS OR IMPLIED
#  WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF
#  MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
#
# @(#) $Header: 2002/10/09 15:34:11

#
# Various configurable paths (remember to edit Makefile.in, not Makefile)
#

# Top level hierarchy
prefix = @prefix@
# Pathname of directory to install the binary
BINDEST = @prefix@/bin
# Pathname of directory to install the man page
MANDEST = @prefix@/man

BLANK = # make a blank space.  DO NOT add anything to this line

# The following will be redefined under Windows (see WIN32 lable below)
CC = @CC@
CPP = @CXX@
LINK = $(CPP)
MKDEP = ./conf/mkdep
```

```
TCLSH = @V_TCLSH@
TCL2C = @V_TCL2CPP@
AR = ar rc $(BLANK)

RANLIB = @V_RANLIB@
INSTALL = @INSTALL@
LN = ln
TEST = test
RM = rm -f
MV      = mv
PERL = @PERL@

# for diffusion
#DIFF_INCLUDES = "./diffusion3/main ./diffusion3/lib
# ./diffusion3/nr ./diffusion3/ns"

# Flags for creating a shared library - IANS Additions

SHARED = -dynamiclib -lresolv

CCOPT = @V_CCOPT@
STATIC = @V_STATIC@
LDFLAGS = $(STATIC)
LDOUT = -o $(BLANK)

############################ Protolib Section ######################

PROTOLIB = ../../protolib

PROTOLIB_INCLUDES = -I$(PROTOLIB)/common -I$(PROTOLIB)/ns
PROTOLIB_FLAGS = -DUNIX -DNS2 -DPROTO_DEBUG -DHAVE_ASSERT

OBJ_PROTOLIB_CPP = \
            $(PROTOLIB)/ns/nsProtoAgent.o $(PROTOLIB)/common/protoSim.o \
            $(PROTOLIB)/common/networkAddress.o \
            $(PROTOLIB)/common/protocolTimer.o \
            $(PROTOLIB)/common/debug.o

############################## PAI Section #########################

PAI = ../../../Apps/nrl/p2ps-ns2/src/pai

PAI_CORE = $(PAI)/core
PAI_FACTORY = $(PAI)/factory
PAI_IMPL = $(PAI)/impl
PAI_JNI = $(PAI)/jni

#Note just include the ns implementation here - NOT the net directory
```

```
PAI_LIB = -framework JavaVM
PAI_SHARED_LDFLAGS = $(SHARED)

PAI_INCLUDES = -I$(JAVA_HOME)/include -I$(PAI_CORE)
 -I$(PAI_FACTORY) -I$(PAI_IMPL) -I$(PAI_IMPL)/ns -I$(PAI_JNI)

OBJ_PAI_CPP = $(PAI_CORE)/LinkedList.o $(PAI_IMPL)/PAIDispatcher.o \
$(PAI_IMPL)/PAIEngine.o $(PAI_FACTORY)/PAIFactory.o \
$(PAI_CORE)/PAIMultipleListener.o $(PAI_IMPL)/PAISocket.o \
$(PAI_IMPL)/PAITimer.o $(PAI_IMPL)/ns/PAINS2UDPSocket.o \
$(PAI_IMPL)/ns/PAINS2Timer.o $(PAI_CORE)/PAIListener.o \
$(PAI_CORE)/PAI.o $(PAI_IMPL)/ns/PAIAgent.o\
$(PAI_IMPL)/ns/PAIBroker.o\
$(PAI_FACTORY)/PAIEnvironment.o\
$(PAI_JNI)/JVMRef.o $(PAI_IMPL)/ns/JavaAgent.o\
$(PAI_JNI)/JavaPAI.o $(PAI_JNI)/JavaEnv.o


######################### END PAI Section ############################

DEFINE = -DTCP_DELAY_BIND_ALL -DNO_TK @V_DEFINE@
@V_DEFINES@ @DEFS@ -DNS_DIFFUSION -DSMAC_NO_SYNC
-DSTL_NAMESPACE=@STL_NAMESPACE@ -DUSE_SINGLE_ADDRESS_SPACE

INCLUDES = \
$(PROTOLIB_INCLUDES) \
$(PAI_INCLUDES) \
-I. @V_INCLUDE_X11@ \
@V_INCLUDES@ \
-I./tcp -I./common -I./link -I./queue \
-I./adc -I./apps -I./mac -I./mobile -I./trace \
-I./routing -I./tools -I./classifier -I./mcast \
-I./diffusion3/lib/main -I./diffusion3/lib \
-I./diffusion3/lib/nr -I./diffusion3/ns \
-I./diffusion3/diffusion -I./asim/ -I./qs


LIB = \
@V_LIBS@ \
@V_LIB_X11@ \
@V_LIB@ \
$(PAI_LIB) \
-lm @LIBS@
# -L@libdir@ \

CFLAGS = $(CCOPT) $(DEFINE) $(PROTOLIB_FLAGS) $(PAI_FLAGS)

# Explicitly define compilation rules since SunOS 4's make doesn't like gcc.
# Also, gcc does not remove the .o before forking 'as', which can be a
```

```
# problem if you don't own the file but can write to the directory.
.SUFFIXES: .cc # $(.SUFFIXES)

.cc.o:
@rm -f $@
$(CPP) -c $(CFLAGS) $(INCLUDES) -o $@ $*.cc

.c.o:
@rm -f $@
$(CC) -c $(CFLAGS) $(INCLUDES) -o $@ $*.c


GEN_DIR = gen/
LIB_DIR = lib/
NS = ns
NSX = nsx
NSE = nse

# To allow conf/makefile.win overwrite this macro
# We will set these two macros to empty in conf/makefile.win
# since VC6.0
# does not seem to support the STL in gcc 2.8 and up.
OBJ_STL = diffusion3/lib/nr/nr.o diffusion3/lib/dr.o \
diffusion3/ns/diffagent.o diffusion3/ns/diffrtg.o \
diffusion3/ns/difftimer.o \
diffusion3/diffusion/diffusion.o \
diffusion3/lib/main/attrs.o \
diffusion3/lib/main/iodev.o \
diffusion3/lib/main/timers.o \
diffusion3/lib/main/events.o \
diffusion3/lib/main/message.o \
diffusion3/lib/main/stats.o \
diffusion3/lib/main/tools.o \
diffusion3/lib/drivers/rpc_stats.o \
diffusion3/apps/sysfilters/gradient.o \
diffusion3/apps/sysfilters/log.o \
diffusion3/apps/sysfilters/tag.o \
diffusion3/apps/sysfilters/srcrt.o \
diffusion3/lib/diffapp.o \
diffusion3/apps/pingapp/ping_sender.o \
diffusion3/apps/pingapp/ping_receiver.o \
diffusion3/apps/pingapp/ping_common.o \
diffusion3/apps/pingapp/push_receiver.o \
diffusion3/apps/pingapp/push_sender.o \
diffusion3/apps/gear/geo-attr.o \
diffusion3/apps/gear/geo-routing.o \
diffusion3/apps/gear/geo-tools.o \
nix/hdr_nv.o nix/classifier-nix.o \
nix/nixnode.o nix/nixvec.o \
```

```
nix/nixroute.o

NS_TCL_LIB_STL = tcl/lib/ns-diffusion.tcl

# WIN32: uncomment the following line to include specific make for VC++
# !include <conf/makefile.win>

OBJ_CC = \
tools/random.o tools/rng.o tools/ranvar.o common/misc.o\
common/timer-handler.o \
common/scheduler.o common/object.o common/packet.o \
common/ip.o routing/route.o common/connector.o common/ttl.o \
trace/trace.o trace/trace-ip.o \
classifier/classifier.o classifier/classifier-addr.o \
classifier/classifier-hash.o \
classifier/classifier-virtual.o \
classifier/classifier-mcast.o \
classifier/classifier-bst.o \
classifier/classifier-mpath.o mcast/replicator.o \
classifier/classifier-mac.o \
classifier/classifier-qs.o \
classifier/classifier-port.o src_rtg/classifier-sr.o \
        src_rtg/sragent.o src_rtg/hdr_src.o adc/ump.o \
qs/qsagent.o qs/hdr_qs.o \
apps/app.o apps/telnet.o tcp/tcplib-telnet.o \
tools/trafgen.o trace/traffictrace.o tools/pareto.o \
tools/expoo.o tools/cbr_traffic.o \
adc/tbf.o adc/resv.o adc/sa.o tcp/saack.o \
tools/measuremod.o adc/estimator.o adc/adc.o adc/ms-adc.o \
adc/timewindow-est.o adc/acto-adc.o \
        adc/pointsample-est.o adc/salink.o adc/actp-adc.o \
adc/hb-adc.o adc/expavg-est.o\
adc/param-adc.o adc/null-estimator.o \
adc/adaptive-receiver.o apps/vatrcvr.o adc/consrcvr.o \
common/agent.o common/message.o apps/udp.o \
common/session-rtp.o apps/rtp.o tcp/rtcp.o \
common/ivs.o \
tcp/tcp.o tcp/tcp-sink.o tcp/tcp-reno.o \
tcp/tcp-newreno.o \
tcp/tcp-vegas.o tcp/tcp-rbp.o tcp/tcp-full.o tcp/rq.o \
baytcp/tcp-full-bay.o baytcp/ftpc.o baytcp/ftps.o \
tcp/scoreboard.o tcp/scoreboard-rq.o tcp/tcp-sack1.o tcp/tcp-fack.o \
tcp/tcp-asym.o tcp/tcp-asym-sink.o tcp/tcp-fs.o \
tcp/tcp-asym-fs.o tcp/tcp-qs.o \
tcp/tcp-int.o tcp/chost.o tcp/tcp-session.o \
tcp/nilist.o \
tools/integrator.o tools/queue-monitor.o \
tools/flowmon.o tools/loss-monitor.o \
queue/queue.o queue/drop-tail.o \
```

```
adc/simple-intserv-sched.o queue/red.o \
queue/semantic-packetqueue.o queue/semantic-red.o \
tcp/ack-recons.o \
queue/sfq.o queue/fq.o queue/drr.o queue/srr.o queue/cbq.o \
queue/jobs.o queue/marker.o queue/demarker.o \
link/hackloss.o queue/errmodel.o queue/fec.o\
link/delay.o tcp/snoop.o \
gaf/gaf.o \
link/dynalink.o routing/rtProtoDV.o common/net-interface.o \
mcast/ctrMcast.o mcast/mcast_ctrl.o mcast/srm.o \
common/sessionhelper.o queue/delaymodel.o \
mcast/srm-ssm.o mcast/srm-topo.o \
apps/mftp.o apps/mftp_snd.o apps/mftp_rcv.o \
apps/codeword.o \
routing/alloc-address.o routing/address.o \
$(LIB_DIR)int.Vec.o $(LIB_DIR)int.RVec.o \
$(LIB_DIR)dmalloc_support.o \
webcache/http.o webcache/tcp-simple.o webcache/pagepool.o \
webcache/inval-agent.o webcache/tcpapp.o webcache/http-aux.o \
webcache/mcache.o webcache/webtraf.o \
webcache/webserver.o \
webcache/logweb.o \
empweb/empweb.o \
empweb/empftp.o \
realaudio/realaudio.o \
mac/lanRouter.o classifier/filter.o \
common/pkt-counter.o \
common/Decapsulator.o common/Encapsulator.o \
common/encap.o \
mac/channel.o mac/mac.o mac/ll.o mac/mac-802_11.o \
mac/mac-802_3.o mac/mac-tdma.o mac/smac.o \
mobile/mip.o mobile/mip-reg.o mobile/gridkeeper.o \
mobile/propagation.o mobile/tworayground.o \
mobile/antenna.o mobile/omni-antenna.o \
mobile/shadowing.o mobile/shadowing-vis.o mobile/dumb-agent.o \
common/bi-connector.o common/node.o \
common/mobilenode.o \
mac/arp.o mobile/god.o mobile/dem.o \
mobile/topography.o mobile/modulation.o \
queue/priqueue.o queue/dsr-priqueue.o \
mac/phy.o mac/wired-phy.o mac/wireless-phy.o \
mac/mac-timers.o trace/cmu-trace.o mac/varp.o \
dsdv/dsdv.o dsdv/rtable.o queue/rtqueue.o \
routing/rttable.o \
imep/imep.o imep/dest_queue.o imep/imep_api.o \
imep/imep_rt.o imep/rxmit_queue.o imep/imep_timers.o \
imep/imep_util.o imep/imep_io.o \
tora/tora.o tora/tora_api.o tora/tora_dest.o \
tora/tora_io.o tora/tora_logs.o tora/tora_neighbor.o \
```

```
dsr/dsragent.o dsr/hdr_sr.o dsr/mobicache.o dsr/path.o \
dsr/requesttable.o dsr/routecache.o dsr/add_sr.o \
dsr/dsr_proto.o dsr/flowstruct.o dsr/linkcache.o \
dsr/simplecache.o dsr/sr_forwarder.o \
aodv/aodv_logs.o aodv/aodv.o \
aodv/aodv_rtable.o aodv/aodv_rqueue.o \
common/ns-process.o \
satellite/satgeometry.o satellite/sathandoff.o \
satellite/satlink.o satellite/satnode.o \
satellite/satposition.o satellite/satroute.o \
satellite/sattrace.o \
rap/raplist.o rap/rap.o rap/media-app.o rap/utilities.o \
common/fsm.o tcp/tcp-abs.o \
diffusion/diffusion.o diffusion/diff_rate.o diffusion/diff_prob.o \
diffusion/diff_sink.o diffusion/flooding.o diffusion/omni_mcast.o \
diffusion/hash_table.o diffusion/routing_table.o diffusion/iflist.o \
tcp/tfrc.o tcp/tfrc-sink.o mobile/energy-model.o apps/ping.o tcp/tcp-rfc793edu.o \
queue/rio.o queue/semantic-rio.o tcp/tcp-sack-rh.o tcp/scoreboard-rh.o \
plm/loss-monitor-plm.o plm/cbr-traffic-PP.o \
linkstate/hdr-ls.o \
mpls/classifier-addr-mpls.o mpls/ldp.o mpls/mpls-module.o \
routing/rtmodule.o classifier/classifier-hier.o \
routing/addr-params.o \
routealgo/rnode.o \
routealgo/bfs.o \
routealgo/rbitmap.o \
routealgo/rlookup.o \
routealgo/routealgo.o \
diffserv/dsred.o diffserv/dsredq.o \
diffserv/dsEdge.o diffserv/dsCore.o \
diffserv/dsPolicy.o diffserv/ew.o\
queue/red-pd.o queue/pi.o queue/vq.o queue/rem.o \
queue/gk.o \
pushback/rate-limit.o pushback/rate-limit-strategy.o \
pushback/ident-tree.o pushback/agg-spec.o \
pushback/logging-data-struct.o \
pushback/rate-estimator.o \
pushback/pushback-queue.o pushback/pushback.o \
common/parentnode.o trace/basetrace.o \
common/simulator.o asim/asim.o \
common/scheduler-map.o common/splay-scheduler.o \
linkstate/ls.o linkstate/rtProtoLS.o \
pgm/classifier-pgm.o pgm/pgm-agent.o pgm/pgm-sender.o \
pgm/pgm-receiver.o mcast/rcvbuf.o \
mcast/classifier-lms.o mcast/lms-agent.o mcast/lms-receiver.o \
mcast/lms-sender.o \
@V_STLOBJ@
```

```
# don't allow comments to follow continuation lines

#  mac-csma.o mac-multihop.o\
# sensor-nets/landmark.o mac-simple-wireless.o \
# sensor-nets/tags.o sensor-nets/sensor-query.o \
# sensor-nets/flood-agent.o \

# what was here before is now in emulate/
OBJ_C =

OBJ_COMPAT = $(OBJ_GETOPT) common/win32.o
#XXX compat/win32x.o compat/tkConsole.o

OBJ_EMULATE_CC = \
emulate/net-ip.o \
emulate/net.o \
emulate/tap.o \
emulate/ether.o \
emulate/internet.o \
emulate/ping_responder.o \
emulate/arp.o \
emulate/icmp.o \
emulate/net-pcap.o \
emulate/nat.o  \
emulate/iptap.o \
emulate/tcptap.o

OBJ_EMULATE_C = \
emulate/inet.o

OBJ_GEN = $(GEN_DIR)version.o $(GEN_DIR)ns_tcl.o $(GEN_DIR)ptypes.o

SRC = $(OBJ_C:.o=.c) $(OBJ_CC:.o=.cc) \
$(OBJ_EMULATE_C:.o=.c) $(OBJ_EMULATE_CC:.o=.cc) \
$(OBJ_CPP:.o=.cpp) \
common/tclAppInit.cc common/tkAppInit.cc

OBJ = $(OBJ_C) $(OBJ_CC) $(OBJ_GEN) $(OBJ_COMPAT)
$(OBJ_PROTOLIB_CPP) $(OBJ_PAI_CPP)

CLEANFILES = ns nse nsx ns.dyn $(OBJ) $(OBJ_EMULATE_CC) \
$(OBJ_EMULATE_C) common/tclAppInit.o \
$(GEN_DIR)* $(NS).core core core.$(NS) core.$(NSX) core.$(NSE) \
common/ptypes2tcl common/ptypes2tcl.o

SUBDIRS=\
indep-utils/cmu-scen-gen/setdest \
indep-utils/webtrace-conv/dec \
indep-utils/webtrace-conv/epa \
```

```
indep-utils/webtrace-conv/nlanr \
indep-utils/webtrace-conv/ucb

BUILD_NSE = @build_nse@

all: $(NS) $(BUILD_NSE) all-recursive


all-recursive:
for i in $(SUBDIRS); do ( cd $$i; $(MAKE) all; ) done

$(NS): $(OBJ) common/tclAppInit.o Makefile
$(LINK) $(LDFLAGS) $(LDOUT)$@ \
common/tclAppInit.o $(OBJ) $(LIB)

Makefile: Makefile.in
@echo "Makefile.in is newer than Makefile."
@echo "You need to re-run configure."
false

$(NSE): $(OBJ) common/tclAppInit.o $(OBJ_EMULATE_CC) $(OBJ_EMULATE_C)
$(LINK) $(LDFLAGS) $(LDOUT)$@ \
common/tclAppInit.o $(OBJ) \
$(OBJ_EMULATE_CC) $(OBJ_EMULATE_C)  $(LIB)

ns.dyn: $(OBJ) common/tclAppInit.o
$(LINK) $(LDFLAGS) -o $@ \
common/tclAppInit.o $(OBJ) $(LIB)

libnspai.dyn: $(OBJ) common/tclAppInit.o
$(LINK) $(PAI_SHARED_LDFLAGS) -o $@ \
common/tclAppInit.o $(OBJ) $(LIB)

PURIFY = purify -cache-dir=/tmp
ns-pure: $(OBJ) common/tclAppInit.o
$(PURIFY) $(LINK) $(LDFLAGS) -o $@ \
common/tclAppInit.o $(OBJ) $(LIB)

NS_TCL_LIB = \
tcl/lib/ns-compat.tcl \
tcl/lib/ns-default.tcl \
tcl/lib/ns-errmodel.tcl \
tcl/lib/ns-lib.tcl \
tcl/lib/ns-link.tcl \
tcl/lib/ns-mobilenode.tcl \
tcl/lib/ns-sat.tcl \
tcl/lib/ns-cmutrace.tcl \
tcl/lib/ns-node.tcl \
tcl/lib/ns-rtmodule.tcl \
```

```
tcl/lib/ns-hiernode.tcl \
tcl/lib/ns-packet.tcl \
tcl/lib/ns-queue.tcl \
tcl/lib/ns-source.tcl \
tcl/lib/ns-nam.tcl \
tcl/lib/ns-trace.tcl \
tcl/lib/ns-agent.tcl \
tcl/lib/ns-random.tcl \
tcl/lib/ns-namsupp.tcl \
tcl/lib/ns-address.tcl \
tcl/lib/ns-intserv.tcl \
tcl/lib/ns-autoconf.tcl \
tcl/rtp/session-rtp.tcl \
tcl/lib/ns-mip.tcl \
tcl/rtglib/dynamics.tcl \
tcl/rtglib/route-proto.tcl \
tcl/rtglib/algo-route-proto.tcl \
tcl/rtglib/ns-rtProtoLS.tcl \
        tcl/interface/ns-iface.tcl \
tcl/mcast/BST.tcl \
        tcl/mcast/ns-mcast.tcl \
        tcl/mcast/McastProto.tcl \
        tcl/mcast/DM.tcl \
tcl/mcast/srm.tcl \
tcl/mcast/srm-adaptive.tcl \
tcl/mcast/srm-ssm.tcl \
tcl/mcast/timer.tcl \
tcl/mcast/McastMonitor.tcl \
tcl/mcast/mftp_snd.tcl \
tcl/mcast/mftp_rcv.tcl \
tcl/mcast/mftp_rcv_stat.tcl \
tcl/mobility/dsdv.tcl \
tcl/mobility/dsr.tcl \
        tcl/ctr-mcast/CtrMcast.tcl \
        tcl/ctr-mcast/CtrMcastComp.tcl \
        tcl/ctr-mcast/CtrRPComp.tcl \
tcl/rlm/rlm.tcl \
tcl/rlm/rlm-ns.tcl \
tcl/session/session.tcl \
tcl/lib/ns-route.tcl \
tcl/emulate/ns-emulate.tcl \
tcl/lan/vlan.tcl \
tcl/lan/abslan.tcl \
tcl/lan/ns-ll.tcl \
tcl/lan/ns-mac.tcl \
tcl/webcache/http-agent.tcl \
tcl/webcache/http-server.tcl \
tcl/webcache/http-cache.tcl \
tcl/webcache/http-mcache.tcl \
```

```
tcl/webcache/webtraf.tcl \
tcl/webcache/empweb.tcl \
tcl/webcache/empftp.tcl \
tcl/plm/plm.tcl \
tcl/plm/plm-ns.tcl \
tcl/plm/plm-topo.tcl \
tcl/mpls/ns-mpls-classifier.tcl \
tcl/mpls/ns-mpls-ldpagent.tcl \
tcl/mpls/ns-mpls-node.tcl \
tcl/mpls/ns-mpls-simulator.tcl \
tcl/lib/ns-pushback.tcl \
tcl/lib/ns-srcrt.tcl \
tcl/mcast/ns-lms.tcl \
tcl/lib/ns-qsnode.tcl \
@V_NS_TCL_LIB_STL@

$(GEN_DIR)ns_tcl.cc: $(NS_TCL_LIB)
$(TCLSH) bin/tcl-expand.tcl tcl/lib/ns-lib.tcl @V_NS_TCL_LIB_STL@
| $(TCL2C) et_ns_lib > $@

$(GEN_DIR)version.c: VERSION
$(RM) $@
$(TCLSH) bin/string2c.tcl version_string < VERSION > $@

$(GEN_DIR)ptypes.cc: common/ptypes2tcl common/packet.h
./common/ptypes2tcl > $@

common/ptypes2tcl: common/ptypes2tcl.o
$(LINK) $(LDFLAGS) $(LDOUT)$@ common/ptypes2tcl.o

common/ptypes2tcl.o: common/ptypes2tcl.cc common/packet.h

install: force install-ns install-man install-recursive

install-ns: force
$(INSTALL) -m 555 -o bin -g bin ns $(DESTDIR)$(BINDEST)

install-man: force
$(INSTALL) -m 444 -o bin -g bin ns.1 $(DESTDIR)$(MANDEST)/man1

install-recursive: force
for i in $(SUBDIRS); do ( cd $$i; $(MAKE) install; ) done

clean:
$(RM) $(CLEANFILES)

AUTOCONF_GEN = tcl/lib/ns-autoconf.tcl
distclean: distclean-recursive
$(RM) $(CLEANFILES) Makefile config.cache config.log config.status \
```

```
    autoconf.h gnuc.h os-proto.h $(AUTOCONF_GEN); \
$(MV) .configure .configure- ;\
echo "Moved .configure to .configure-"

distclean-recursive:
for i in $(SUBDIRS); do ( cd $$i; $(MAKE) clean; $(RM) Makefile; ) done

tags: force
ctags -wtd *.cc *.h webcache/*.cc webcache/*.h dsdv/*.cc dsdv/*.h \
dsr/*.cc dsr/*.h webcache/*.cc webcache/*.h lib/*.cc lib/*.h \
../Tcl/*.cc ../Tcl/*.h

TAGS: force
etags *.cc *.h webcache/*.cc webcache/*.h dsdv/*.cc dsdv/*.h \
dsr/*.cc dsr/*.h webcache/*.cc webcache/*.h lib/*.cc lib/*.h \
../Tcl/*.cc ../Tcl/*.h

tcl/lib/TAGS: force
( \
cd tcl/lib; \
$(TCLSH) ../../bin/tcl-expand.tcl ns-lib.tcl | grep '^### tcl-expand.tcl:
begin' | awk '{print $$5}' >.tcl_files; \
etags --lang=none -r '/^[ \t]*proc[ \t]+\([^ \t]+\)/\1/' `cat .tcl_files`; \
etags --append --lang=none -r '/^\([A-Z][^ \t]+\)[ \t]+
\(instproc\|proc\)[ \t]+\([^ \t]+\)[ \t]+/\1::\3/' `cat .tcl_files`; \
)

depend: $(SRC)
$(MKDEP) $(CFLAGS) $(INCLUDES) $(SRC)

srctar:
@cwd=`pwd` ; dir=`basename $$cwd` ; \
    name=ns-`cat VERSION | tr A-Z a-z` ; \
    tar=ns-src-`cat VERSION`.tar.gz ; \
    list="" ; \
    for i in `cat FILES` ; do list="$$list $$name/$$i" ; done; \
    echo \
    "(rm -f $$tar; cd .. ; ln -s $$dir $$name)" ; \
     (rm -f $$tar; cd .. ; ln -s $$dir $$name) ; \
    echo \
    "(cd .. ; tar cfh $$tar [lots of files])" ; \
     (cd .. ; tar cfh - $$list) | gzip -c > $$tar ; \
    echo \
    "rm ../$$name; chmod 444 $$tar" ;  \
     rm ../$$name; chmod 444 $$tar

force:

test: force
```

```
./validate

.cpp.o:
@rm -f $@
$(CC) -c $(CFLAGS) $(INCLUDES) -o $@ $*.cpp


# Create makefile.vc for Win32 development by replacing:
# "# !include ..."  ->  "!include ..."
makefile.vc: Makefile.in
$(PERL) bin/gen-vcmake.pl < Makefile.in > makefile.vc
# $(PERL) -pe 's/^# (\!include)/\!include/o' < Makefile.in > makefile.vc
```

2.2.2 What's included in the P2PS-NS2 Binding?

The P2PS-NS2 distribution consiststs of several co-operating software stacks, which are described in the following chapters of the manual. It includes the PAI interface to Protolib (described in Chapt. 4, the Java NS2 agent extensions, described in Chapt. 5 and a customised version of P2PS, described in Chapt. 7, along with the necessary *jar* files it needs e.g. jdom.

P2PS had to be de-threaded for this implementation. Therefore, several classes within P2PS had to be modified in order to take out the threading dependencies which are at the core of the P2PS implementation. P2PS was written as a multi-threaded toolkit but this does not fit well with scalability and timing issues for running NS2 simulations. The major classes, which were de-threaded were the UDPInputMontor, and the asynchronous discovery mechanisms, which set of listeners within the background for notification. Within the NS2 implementation, it is assumes that things turn up the a particular order.

2.3 Configuration

The location of the NS2 shared library must be set within the Java source code. At the time of writing, I could not get the *LD_LIBRARY_PATH* working with loading shared libraries from Java on my Mac OS 10.x system. Further, passing this location through to Java is also non-trivial since we are dynamically creating a JVM and therefore this would have to be first passed through the C++ code, then onto the the JVM created at run-time. Therefore, until the LD_LIBRARY_PATH mechanism problem is solved, it is far easier to modify the source and recompile this one Java class, which is easy and quick.

This is set within *PAINative* class, which is in the *pai.impl* package.

```
package pai.impl;

public class PAINative implements PAIInterface, PTIInterface {

....

    static {
        if (Logging.isEnabled())
            System.out.println("PAINative: Java library path = " + System.getProperty("jav
        // System.loadLibrary("PAI");
        // System.load("/Users/scmijt/Apps/nrl/p2ps-ns2/lib/libPAI.so");
        System.load("/Users/scmijt/Install/ns-allinone-2.26/ns-2.26/libnspai.dyn");
        //System.loadLibrary("nspai");
    }
```

You need to replace the following path:

```
System.load("/Users/scmijt/Install/ns-allinone-2.26/ns-2.26/libnspai.dyn");
```

to:

```
System.load("YOUR_NS_ROOT/ns-2.26/libnspai.dyn");
```

and then recompile this class by cd'ing into the P2PS installation directory, and using the following commands:

```
cd src/jpai/pai/impl/
javac -classpath ../../../../classes -d ../../../../classes PAINative.java
```

which will compile the class and put it in the correct place.

2.4 conclusion

This chapter described the installation of the core packages needed for P2PS-NS2. The Protolib library needs to be installed first, followed by the P2PS-NS2 installation. Both installation require the editing of the NS Makefile.in file in order to add the correct dependencies into NS2. P2PS-NS requires the installation of both the static and shared libraries for the NS2 executable and the JNI bindings describes later in this manual.

Chapter 3

Protolib

3.1 An overview of Protolib

A typical usage scenario of Protolib is given in the Fig 3.2 . Here, a timer is set up to trigger every 100 milliseconds. Upon a trigger event, a C++ callback is invoked that allows the application developer to integrate an event action. In this example, the application sends a UDP message using Protolib. This communication mechanism can be achieved using a standard UDP call across a network or between two NS-2 nodes. When the packet is received by the receiver, another event is generates indicating that data has been received. This, in turn, calls a routine that allows the application to collect the data from the UDP port and process it in some way. The application interface between the Protolib and the P2P middleware abstracts the reliance on specific networking/timing mechanisms in Protolib to create a generalized pluggable transport mechanism. Within PAI, middleware or indeed applications program to one interface and then choose the environment they want to run within e.g. Network or NS-2. This is very similar to GATLite but at a far lower level. PAI also support multiple sockets, timers and corresponding listeners for timeouts or UDP receive data events and establishes a cooperating event dispatching mechanism using multithreading.

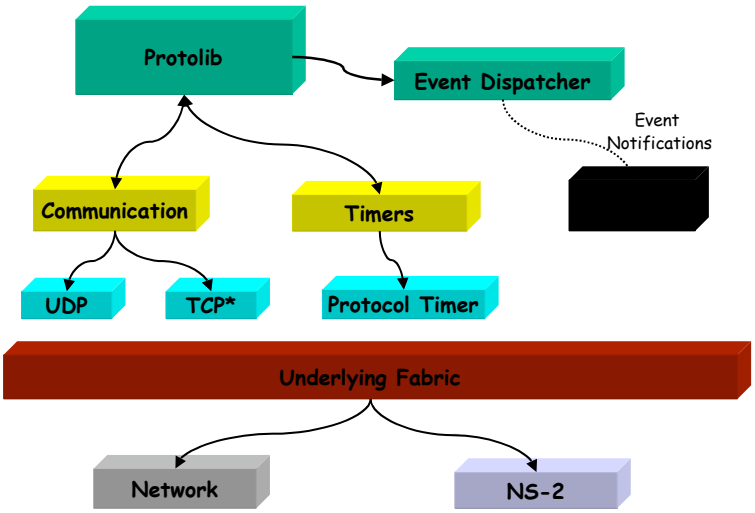

Figure 3.1: An overview of the Protolib toolkit, showing the three distinct components, sockets timers and a mechanism for dispatching events.

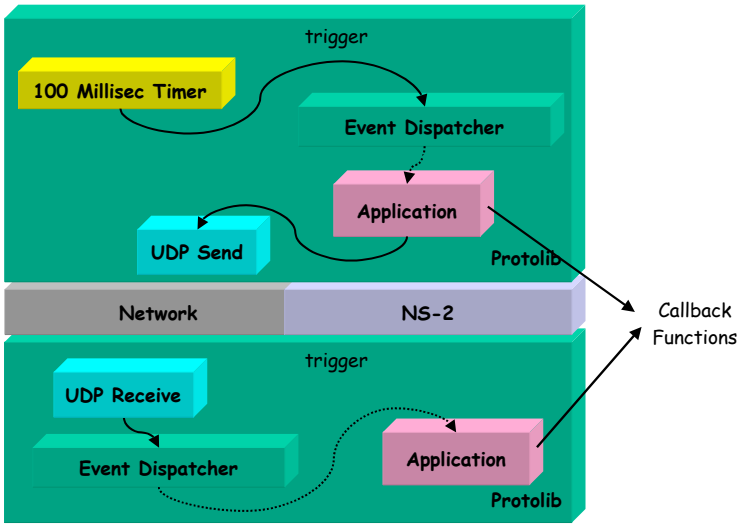

Figure 3.2: An overview of the functionality provided by the ProtoApp application, which triggers a data send once-per-second.

Chapter 4

The PAI Interface

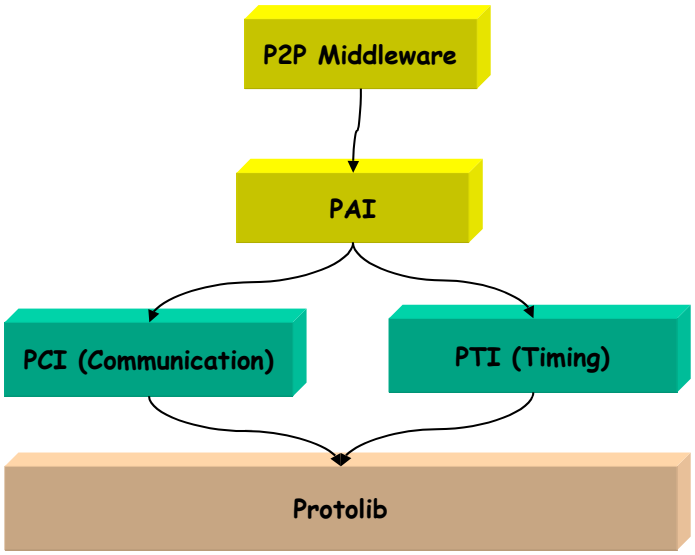

Figure 4.1: An overview of the PAI interface, showing the two sections to the underlying Protolib sockets and timers.

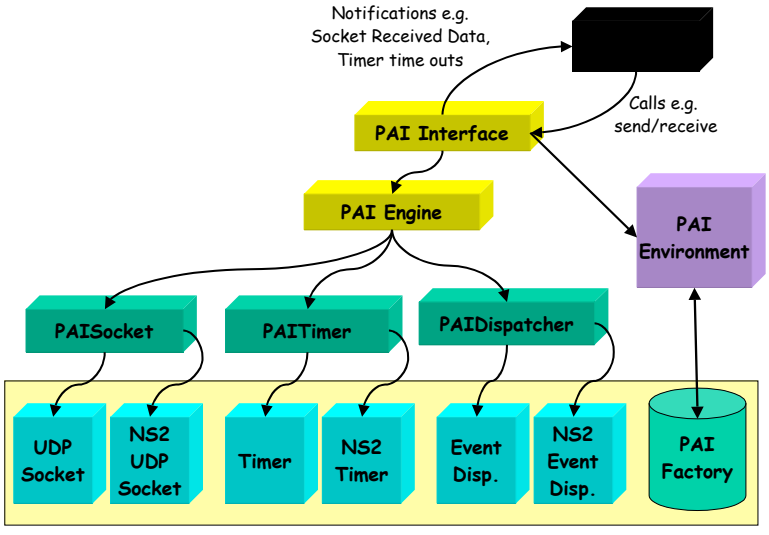

Figure 4.2: The PAI interface uses the Factory design pattern to create a common high level interface to whatever sockets or underlying timers the programmer is using.

When Timer times out:

```
void PAI_Example::OnTxTimeout() {
.....
    pci->send(sock1, "127.0.0.1", buffer, len);
}
```

When Data is Received:

```
void PAI_Example::OnSocketRecv() {
 .....
    char *buf = pci->recv(sock1, &addr, &len);
}
```

Example Main Program:

```
pai.getEnvironment()->setBinding(PAI_NETWORK);
pai.getEnvironment()->setNetworkProtocol(PAI_UDP);

timer = pti->addTimer(1.0, 5);
sock = pci->addSocket(5004);

pci->addListener(sock, this, (CallbackFunc)&PAI_Example::OnTxTimeout);
pti->addListener(timer, this, (CallbackFunc)&PAI_Example::OnSocketRecv);

pti->runTimers();
```

Figure 4.3: An PAI code example, showing how you would implemented the standard Protolib demonstration, which sets of a 1 second timer and sends data between two nodes.

Chapter 5

Java NS2 Agents

5.1 NS-2 Node JNI Integration

Each NS2 node can optionally use Java code within the simulation, which can, in turn, be used to invoke 3rd party Java mechanisms, including the supported P2PS middleware, discussed in the next chapter. This chapter gives an overview of the interaction between the TCL scripts, the C++ NS2 agents and Java objects, which can be accessed from each NS2 node. The various code snippets are taken from the P2PS-NS2 source tree and pointers are referenced relative to the installation directory, when provided.

5.2 NS2 Java Overview

The central class, which implements the bridge between the NS2 nodes and an associated Java object is *JavaAgent*. *JavaAgent* inherits from Protolib's *NsProtoAgent*, which is a C++ NS2 agent that provides the data transport implementation within NS2. By inheriting from this class, *JavaAgent* can use this transport mechanism to transmit UDP packets between NS-2 nodes. However, *JavaAgent* does not use this Protolib class directly. The communication proceeds through two different layers. Firstly, if *JavaAgent* wishes to send packets itself, then it uses the PAI interface, which in turn talks to the Protolib NS2 implementation. If if wishes to create a Java Object and let that send messages, then it uses the Java JNI interface to PAI, which in turn uses the PAI interface to send the actual data packets.

Figure 5.1 shows an overview of this process. Also, shown in this figure is a general overview of how Java fits into the picture. Briefly, each Ns2 *JavaAgent* interfaces through a singleton C++ class (called JVMRef) that understands how create a Java Virtual Machine (JVM) and how to use JNI

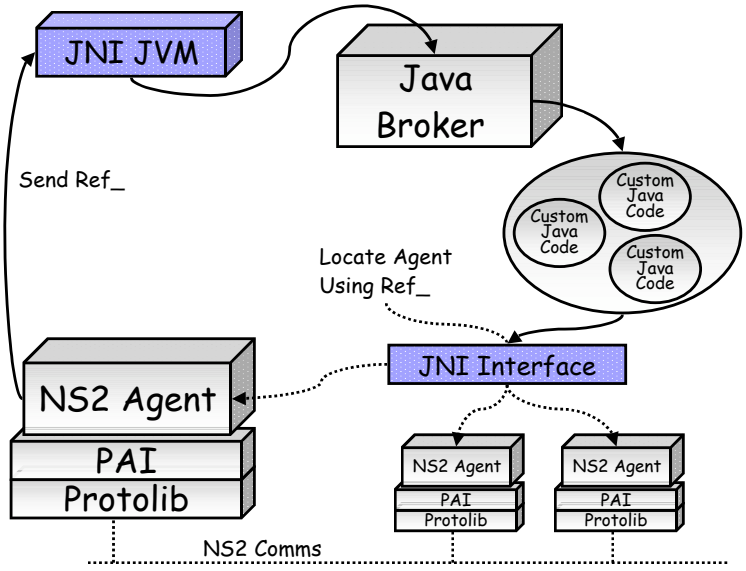

Figure 5.1: An overview of how NS2 interacts with Java

to invoke external Java functions. The *JavaAgent* uses this class to issue commands to the *JavaBroker* class on the Java side in order to create and manage remote objects that are attached to the various Ns2 nodes being created within the simulation.

There could potentially be thousands of NS2 nodes and each one might want to instantiate and use a Java object. Therefore scalability issues can be encountered if this interaction is not slimline enough. Here, the C++ JVM helper class only allows **ONE** JVM to be created no matter how many nodes exist in the simulations. It then uses the *JavaBroker* class, which creates and manages the external Java objects. *JavaBroker* contains functionality that can dynamically create a Java object from a textual representation of its name (e.g. pai.examples.ns2.SimpleCommand). Once created such objects are added to a local Hashtable, which associates an NS2 agent's *ID* with the associated object that has been created for this interaction. The NS2 agent's *ID* is actually its C++ pointer, which is reused later within the JNI binding (see below). Therefore, each NS2 node only instantiates the Java class it needs rather than any other wrapping classes. This implementation therefore maps one-to-one between the C++ NS2 agent and its corresponding Java object and therefore keeps the memory allocation to an absolute minimum.

Each Java class can also interact with other NS2 nodes by using the Java *PAIInterface* or *PTIInterface* interfaces classes within the PAI JNI imple-

mentation. These implementations allow the NS2 Java code to issue commands to send data between NS2 nodes. To do this, a JNI bridge is provided between the Java PAI interface and the corresponding C++ PAI interface that contains the necessary underlying functionality.

The NS2 agent's *ID* is used within the JNI binding to the PAI interface in order to re-associate the Java object within its C++ agent once we are back in the c++ domain i.e. the Java classes need to be able to locate the actual Ns2 agent (in C++) that created this object because they have to send data from their node rather than any node. We therefore need to maintain a reference and pass this through the Java JVM and the JNI PAI interface in order for the the Java implementation route the call appropriately.

5.3 Invoking the Virtual Machine: the C++ to Java Bridge

This section gives a brief overview of the how the NS2 C++ agent (JavaBroker class) invokes a Java virtual machine and creates and maintains references to Java objects for each Ns2 node.

There are three main class in this integration:

1. **JavaAgent.cpp:** the C++ *JavaAgent* class is an NS2 agent that understands how to create and interact with the associated Java objects for this NS2 node. It uses the *JVMRef* class to create a reference to a single Java virtual machine and thereafter uses the and the *JavaBroker* class to locate and instantiate Java objects.

2. **JVMRef.cpp:** This class creates a Java virtual machine and maintains a single reference to this. Only one virtual machine can be instantiated using this class. It also contains a number of convenience methods, which hide the JNI details for interacting with the *JavaBroker* class and provides a clean interface for the C++ *JavaAgent* to use.

3. **JavaBroker.java:** this class allows the C++ *JavaBroker* to create Java objects and provides a container for these objects during the lifetime of the simulation. A Java Hashtable is used to store each Java object along with its identifier, which is the reference to the NS2 agent that this Java object belongs to. Therefore, each NS2 agent can instantiate only one Java class and there is a one-to-one interaction between an NS2 agent and its Java object.

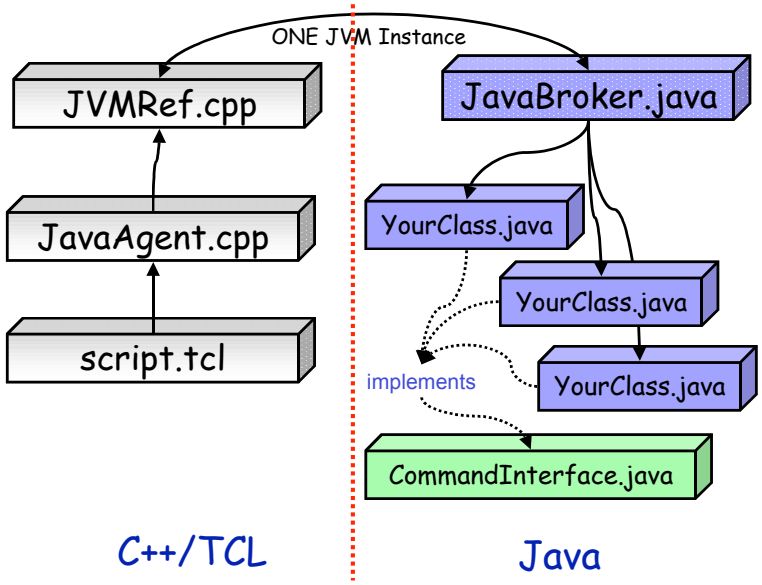

Figure 5.2: The C++ and Java classes used to implement the TCL/C++ and Java bridging mechanism.

These interactions are shown in detail in Figure 5.2. Briefly, the programmer sets the classpath and the actual java class being created in the TCL script. The JavaAgent then passes this data via the JVMRef class to the JavaBroker Java object. The JavaBroker object creates this object on-the-fly from the supplied name. This means that you can access multiple java objects of the same type or different types for different nodes, depending on what you want to implement. For example, you could have a Java data collector agent talking to a Java data collection manager node instance. The only stipulation on the Java objects being created is that they should implement the *CommandInterface*.

The java implementation keeps track of each object in a Hashtable and contains various housekeeping methods for the control of these objects.

5.4 Invoking Java Agents from NS2 Agents

Figure 5.3 shows interaction between an agent and its associated Java class. The programmer who wishes to use this Java functionality within their NS2 simulations only needs to be concerned within their NS2 TCL script and their Java class that implements the behaviour. The relationship between

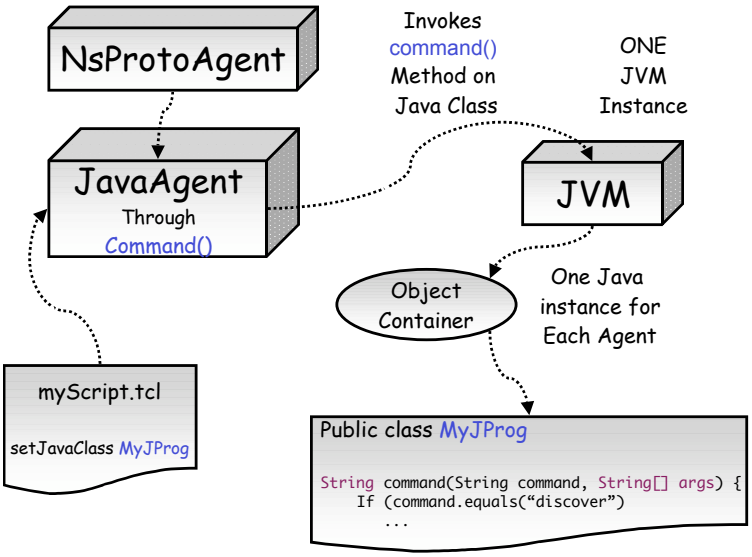

Figure 5.3: The interface to a Java program for an agent employs a similar interface to that of NS2 when communicating between the TCL scripts and the C++ classes.

an Ns2 agent and its Java class is very similar to the relationship between an NS2 TCL script and its associated C++ class (i.e. an NS2 agent) which implements the same kind of interaction through sending text commands between the two. The Java interface employs the same mechanism to bridge these different programming languages. The C++ agent (JavaAgent) simply acts as a go-between and passes that commands across to the appropriate Java object.

Therefore, the interface between the NS2 *JavaAgent* and the chosen Java Class it will interact with, uses the same command-style interface as the TCL-C++ interface for invoking functionality on NS2 agents. This command-style interaction satisfies some essential constraints:

- **Flexibility:** it will keep the flexibility of being able to use NS2 agents in any way programmer sees fit - the Java extensions are optional and any agent extending the *JavaAgent* can choose to use this functionality. However, the core C++ agent code can be programmed to incorporate and other functionality needed beyond the scope of Java.

- **Simplicity:** the scalability issues and framework for interacting with the Java objects can easily be hidden behind the container C++ and

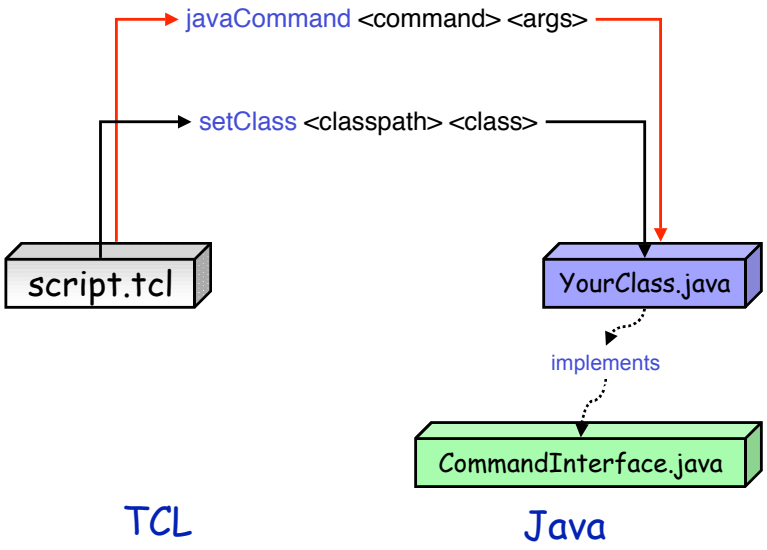

Figure 5.4: The user's view of the interaction between the NS2 agent/script and the Java class associated with that NS2 node.

Java classes - the programmer does not need to be aware of their presence.

- **Familiarity:** this mechanism allows communication between the NS2 agent and any attached Java class through the same familiar interface as NS2 programmers interface between the TCL scripts and C++ agents now.

This interaction is shown in Figure 5.4, which shows some *JavaAgent* commans for specifying and attaching a Java object and for sending it commands. These are the minimum commands needed in order to use your Java object. Each Ns2 node creates a Java object of its own choice by using the TCL command:

```
setClass <classpath> <class>
```

which allows the Java classpath to be set along with the name of the Java class to be instantiated for this NS2 node. Once the Java object has been created, commands can be sent by using the TCL command:

```
javaCommand <command> <args>
```

which would invoke the java command with the associated arguments. There are also other commands implemented that allow you to specify the delimiter to make it easier to chunk your arguments in a flexible way and for creating a trigger mechanism. The following 2 sections illustrate these commands through the use of example TCL and Java codes and the next chapter illustrates how you can extend the Java functionality to use PAI in order to send data between your Java objects through the NS2 subsystem.

5.5 Creating and Attaching a Java Agent

This is a *Hello World* example that demonstrates how to specify the Java classpath and choose a Java class to instantiate and attach to your C++ agent. It then implements a simple *hello* function which is invoke on the Java object.

5.5.1 The TCL Side

The following is the TCL script part of the implementation, which creates two *JavaAgent* NS2 nodes that each create a *SimpleCommand* Java object and then invoke a 'hello' command on that object. This example can be found in examples/pai/javaAgent/startJava.tcl.)

```
puts "Starting..."

# Create a simulator instance
set ns_ [new Simulator ]

# Create two nodes
set n1 [$ns_ node]
set n2 [$ns_ node]

# Put a link between them
$ns_ duplex-link $n1 $n2 64kb 100ms DropTail
$ns_ queue-limit $n1 $n2 100
$ns_ duplex-link-op $n1 $n2 queuePos 0.5
$ns_ duplex-link-op $n1 $n2 orient right

puts "Creating JavaAgent NS2 agents and attach them to the nodes..."
set p1 [new Agent/JavaAgent]
$ns_ attach-agent $n1 $p1

set p2 [new Agent/JavaAgent]
$ns_ attach-agent $n2 $p2
```

```
puts "CREATED OK          ........ ..."

# Initialize each broker telling it what its NS2 address is

puts "In script: Initializing  ..."

$ns_ at 0.0 "$p1 initAgent"
$ns_ at 0.0 "$p2 initAgent"

puts "Setting Java Object to use by each agent ..."

$ns_ at 0.0 "$p1 setClass
/Users/scmijt/Apps/nrl/p2ps-ns2/classes pai.examples.ns2.SimpleCommand"

$ns_ at 0.0 "$p2 setClass
/Users/scmijt/Apps/nrl/p2ps-ns2/classes pai.examples.ns2.SimpleCommand"

# send a message to each agent and tell it to print it to the screen
# This is a "HelloWorld" program for JavaAgents

$ns_ at 0.0 "$p1 javaCommand hello AStringFromP1"
$ns_ at 0.0 "$p2 javaCommand hello AStringFromP2"

$ns_ at 10.0 "finish $ns_"

proc finish {ns_} {
$ns_ halt
delete $ns_
}

$ns_ run
```

The Java agent parts can be seen in this example. The *setClass* function sets the classpath to the *p2ps-ns2* installations classes directory. Here, the actual java class that this node will be using is specified as *pai.examples.ns2.SimpleCommand*. Note here that you can load in classes that are contained in any java package that you wish as long as you follow the Java conventions for locating the compiled versions of these classes.

Once the Java classes have been located, you can then execute various commands by using the *javaCommand* instruction. Here we ask the Java class to execute a *hello* command and pass a string as an argument, identifying the node that is sending the message i.e. *AStringFromP1*. This simple example demonstrates that two Java objects have been created, one for each node and each Java object has been correctly associated or bound to the particular NS2 node.

5.5.2 The Java Side

On the java side of things each object you want to talk to must implement a standard interface called "CommandInterface" which enforces that every Java object implementing this interface implements this command method:

```
package pai.broker;

public interface CommandInterface {

    public String command(String command, String value);
}
```

Every class that you wish to be used from an NS2 agent must implement this Java interface so that it can understand the instructions that are sent to it. Below, an example Java class is given to illustrate the code involved in this process (the actual Java code for this and all other examples
can be found in the *src/jpai/pai/examples/ns2* directory):

```
package pai.examples.ns2;

import pai.broker.CommandInterface;

public class SimpleCommand implements CommandInterface {

    static int count=0;

    int myID;

    public SimpleCommand() {
        ++count;
        myID=count;
    }

    public String command(String command, String args[]) {

        if (command.equals("hello"))
            System.out.println("SimpleCommand(" + myID + ")
             called with Val: " + args[0]);

        return "All called ok from node " + myID;
    }
}
```

As you can see, this is extremely simple, the C++ and Java JVM class take care of all the complexity. In the command method, you can implement

any behaviour you want. You can also return a *String* to your C++ program as indicited. This could allow you, for example, to discover other NS2 nodes using P2PS and then return their address to your C++ agent and keep the control at this point (helpful for non-java programmers!).

5.6 Changing the Command Delimiter

This example demonstrates how you would change the delimiter used to separate command arguments sent to your Java application. The default is to use a white space (as in NS2) to automatically parse the arguments and send them as a sequence of arguments to your agent or Java object. Within the Java NS2 implementation however, this choice is left up to the programmer. Therefore, you could specify for example a '-' symbol as a delimeter and a sequence such as this

```
8 - cherry apple oranges - to eat
```

would be parsed and sent to you program as 3 strings:

```
8
cherry apple oranges
to eat
```

This allows more flexibility in the way you send instructions to your Java code because it does not limit the input to contiguous strings. The example given below demonstrates how this is achieved from the TCL and Java sides.

5.6.1 The TCL Side

The following is the TCL script part of the implementation, which creates two *JavaAgent* NS2 nodes that each create a *ChangeDelimiter* Java object and then change the delimiter of one of the nodes in order to split up the input with respect to a '-' symbol. Note that setting delimiters is a global process and therefore can be set through any node and will be applied to all nodes. This example can be found in examples/pai/javaAgent/changeDelimiter.tcl)

```
puts "Starting..."

# Create simulator instance
set ns_ [new Simulator]

# Create two nodes
```

```
set n1 [$ns_ node]
set n2 [$ns_ node]

# Put a link between them
$ns_ duplex-link $n1 $n2 64kb 100ms DropTail
$ns_ queue-limit $n1 $n2 100
$ns_ duplex-link-op $n1 $n2 queuePos 0.5
$ns_ duplex-link-op $n1 $n2 orient right

puts "Creating JavaAgent NS2 agents and attach them to the nodes..."
set p1 [new Agent/JavaAgent]
$ns_ attach-agent $n1 $p1

set p2 [new Agent/JavaAgent]
$ns_ attach-agent $n2 $p2

puts "In script: Initializing  ..."

$ns_ at 0.0 "$p1 initAgent"
$ns_ at 0.0 "$p2 initAgent"

puts "Setting Java Object to use by each agent ..."

$ns_ at 0.0 "$p1 setClass
/Users/scmijt/Apps/nrl/p2ps-ns2/classes pai.examples.ns2.ChangeDelimiter"

$ns_ at 0.0 "$p2 setClass
/Users/scmijt/Apps/nrl/p2ps-ns2/classes pai.examples.ns2.ChangeDelimiter"

# Delimiters are global and can be set through any node

$ns_ at 0.0 "$p1 javaCommand setDelimiter -"

$ns_ at 0.0 "$p2 javaCommand hello A-String-From-P2"

$ns_ at 10.0 "finish $ns_"

proc finish {ns_} {
$ns_ halt
delete $ns_
}

$ns_ run
```

The Java classes are located and instantiate as previous. Now, we can use the *javaCommand setDelimiter* instruction to change the delimiter. We set this to '-' and then send a single contiguous string to node p2 (A-String-From-P2) by using the 'hello' command. Now instead of passing this as a

single string (as you would get in the NS2 C++ binding), you would get 4 separate string send to your program, which can be accessed individually, for example, as:

```
A
String
From
P2
```

5.6.2 The Java Side

On the java side *ChangeDelimiter.java* implements the "CommandInterface" to identify that it can process commands:

```
package pai.examples.ns2;

import pai.broker.CommandInterface;

public class ChangeDelimiter implements CommandInterface {

    static int count=0;

    int myID;

    public ChangeDelimiter() {
        ++count;
        myID=count;
    }

    public String command(String command, String args[]) {

        if (command.equals("hello")) {
            System.out.println("Command has "
                    + args.length + " arguments");
            for (int i=0; i<args.length; ++i) {
                System.out.println("Arg[" + i + "] = " + args[i]);
            }
        }

        return "All called ok from node " + myID;
    }
}
```

Here, the 'hello' command simply processes through the arguments and prints each to the screen on a separate line. Therefore, running the script will produce the following output:

```
In script: Initializing  ...
Setting Java Object to use by each agent ...
Classpath is -Djava.class.path=/Users/scmijt/Apps/nrl/p2ps-ns2/classes
command has 4 arguments
Arg[0] = A
Arg[1] = String
Arg[2] = From
Arg[3] = P2
```

5.7 Conclusion

In this chapter, a brief overview of the Java integration was given, from a conceptual perspective and a high-level source-code perspective. Then, two different examples were provided that illustrate how one would attach a Java object to an NS2 node and how one can execute Java commands on that object. Lastly, an example was given that demonstrates how you can change the delimiter used to parse the list of arguments you can send to your Java object. This employs a flexible mechanism that can use any string as a delimiter to send lists or sentences to your Java object without having to parse further.

Chapter 6

Using PAI within Java Agents

In the last chapter, we discussed the way Java objects could be attached to Java agents and invoke from within NS2 simulations. In this chapter, an overview of how such Java nodes can be used to send packages between NS2 nodes by using the PAI interface, described in Chapt. 4. The Java interface contains a an interface to PAI through JNI that enables the Java objects to create sockets, attach listeners to the sockets and trigger events.

6.1 The Java PAI Overview

Figure 6.1 shows an overview of the interaction between the C++ agents, the *JavaBroker* and the Java PAI bridge that enables this to be interfaced with the C++ PAI library. As discussed briefly in the previous chapter, the C++ *JavaAgent* passes the pointer to the C++ agent to the *JavaBroker* when it requests to create and attach a Java agent to the NS2 node.

The *JavaBroker* class uses this pointer to store the created Java object in a hashtable for lookup but also pass this references across to the JNI interface, when a Java object requires the use of the PAI interface. This enables the JNI interface to locate the node that created the Java object and therefore whom is indirectly issuing the commands, which ensures that the data being sent through the sockets is sent from the correct node. The PAI interface sends these commands to the Protolib library, which in turn, uses the Protolib NS2 UDP implementation to send the data between the NS2 nodes.

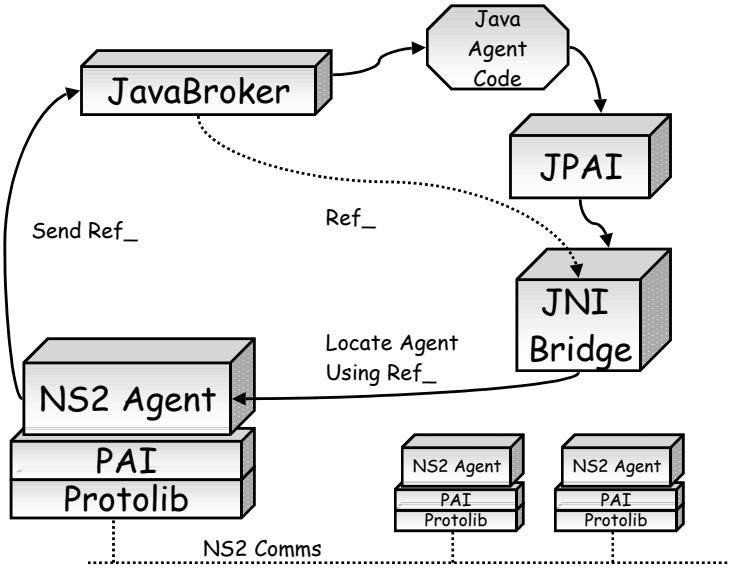

Figure 6.1: An overview PAI is accessed from within a Java node for an NS2 agent

6.2 The Java PAI interface

The Java PAI interface interfaces to the communication part of the PAI interface (i.e. the sockets). It allows the user to create multiple sockets and allows multiple socket listeners to be attached to each socket. This functionality is necessary for the P2PS implementation, described in the next chapter to function correctly.

The Java PAI interface is implemented as a Java interface and uses the *Factory Method Design* pattern [11] in order to create the JNI implementation of this interface. This means that other implementations (e.g. a Java implementation) could be implemented at a later data and trivially plugged in to change the underlying mechanisms used to implement this interface. The application developer however, would not notice this code change because s/he is working with a consistent interface. The JPAI interface can be found in package *pai.api* in the java source tree and is listed below:

```
public interface PAIInterface  {

    void addPAISocketListener(DatagramSocket sock, PAISocketListener listener);
```

```
    void removePAISocketListener(DatagramSocket sock, PAISocketListener listener);

    void open(DatagramSocket sock, int port) throws SocketException;

    DatagramSocket addSocket(int port) throws SocketException;

    void removeSocket(DatagramSocket sock) throws SocketException;

    void setReuseAddress(DatagramSocket sock, boolean on) throws SocketException;

    void setSendBufferSize(DatagramSocket sock, int size) throws SocketException;

    void setReceiveBufferSize(DatagramSocket sock, int size) throws SocketException;

    void setSoTimeout(DatagramSocket sock, int timeout) throws SocketException;

    void send(DatagramSocket sock, DatagramPacket p) throws IOException;

    void receive(DatagramSocket sock, DatagramPacket p) throws IOException;

    void close(DatagramSocket sock);

    void joinGroup(MulticastSocket sock, InetAddress mcastaddr) throws IOException;

    void leaveGroup(MulticastSocket sock, InetAddress mcastaddr) throws IOException;

    void setMulticast(MulticastSocket sock, boolean val);

    public InetAddress getByName(String host) throws UnknownHostException;

    public InetAddress getLocalHost();

    public boolean cleanUp();
    public boolean runBlock();
    public boolean runNonBlock();

    public void setNS2Node(String nodeID);
}
```

Most of the calls are self-explanatory. PAI uses the Java conventions for naming the classes e.g. DatagramSocket and MulticastSocket, both found in the java.net package (see [3]). The PAI Java implementation reimplements the methods from these classes in order to use the PAI interface. This enables the PAI interface to provide the functionality but it leaves the Java interface that developers are familiar with the same. Therefore, to create a Java UDP socket, you simply instantiate a DatagramSocket, which in turn invokes PAI

to create a C++ PAI socket, which in turn creates a Protolib socket.

This design carries the whole weight of the P2PS integration that is discussed in the next chapter. To an application, this conventional object interface to creating UDP sockets means that they require little or no modification in order to use this PAI JNI binding here. For example, in order to get P2PS working with this interface, a new resolver was created which copied the current UDP resolver verbatim and simply did a find and replace on *java.net* with *pa.net* to change the package where P2PS looked for the DatagramSocket class i.e. it uses the PAI implementation rather than the Java one. Everything else follows through the various layers. This re-implementation of these base Java classes can be found in the *pai.net* package in the source tree.

The calls in this interface are illustrated in the examples given later in this chapter.

6.2.1 Using the Java PAI Interface in Ns2 Java Objects

Each Java objects that has been attached to an NS2 node must implement the *PAIAccessInterface* given below, which can be found in the pai.broker package within the source tree:

```
package pai.broker;

import pai.api.PAIInterface;

public interface PAIAccessInterface {

    public void setPAI(PAIInterface pai);
}
```

PAIAccessInterface provides a mechanism for the *JavaBroker* object to create a *PAIInterface* object to the JNI PAI implementation and pass this reference to your Java code. You can then use this reference directly to make PAI calls just as you would if you were using PAI directly.

This mechanism managers the creation and deletion of the PAI JNI implementation and sets variables in the JNI before each invocation so that it has the correct reference to the object it is dealing with at that moment. Briefly, the *JavaBeker* only create **one** instance of the PAI JNI implementation. This means that before each call it must set the reference to the actual NS2 node it is about to issue a command to enabling the interface to create the appropriate binding to PAI at the lower levels. This design adds a small overhead to each call but saves a substantial amount of memory since

it efficiently uses one instance of the code rather that one for each node, which would increase memory consumption greatly (i.e. image if you had thousands of nodes).

6.3 Example 1: Sending Data From One Node to Another

This example uses the Java *PAICommands* class to send data between two NS2 nodes. The actual Java code specifies which nodes to communicate with. This simple example demonstrates how Java objects can be attached to an NS2 nodes and used to create sockets and send data between nodes.

6.3.1 The TCL Side

The following is the TCL script part of the implementation, which creates two *JavaAgent* NS2 nodes attaches the PAICommands Java object to them, initializes them and then sends data from the first node to the second by setting the NS 2 address of the second node directly from the script, using the *setSendTo* command.

```
# Create multicast enabled simulator instance
set ns_ [new Simulator]

# Create two nodes
set n1 [$ns_ node]
set n2 [$ns_ node]

# Put a link between them
$ns_ duplex-link $n1 $n2 64kb 100ms DropTail
$ns_ queue-limit $n1 $n2 100
$ns_ duplex-link-op $n1 $n2 queuePos 0.5
$ns_ duplex-link-op $n1 $n2 orient right

puts "Creating JavaAgent NS2 agents and attach them to the nodes..."
set p1 [new Agent/JavaAgent]
$ns_ attach-agent $n1 $p1

set p2 [new Agent/JavaAgent]
$ns_ attach-agent $n2 $p2

puts "In script: Initializing agents  ..."

$ns_ at 0.0 "$p1 initAgent"
$ns_ at 0.0 "$p2 initAgent"
```

```
puts "Setting Java Object to use by each agent ..."

$ns_ at 0.0 "$p1 setClass
/Users/scmijt/Apps/nrl/p2ps-ns2/classes pai.examples.ns2.PAICommands"
$ns_ at 0.0 "$p2 setClass
/Users/scmijt/Apps/nrl/p2ps-ns2/classes pai.examples.ns2.PAICommands"

puts "Starting simulation ..."

$ns_ at 0.0 "$p1 javaCommand init"
$ns_ at 0.0 "$p2 javaCommand init"

$ns_ at 0.0 "$p1 javaCommand setSendTo [$n2 node-addr]"
$ns_ at 0.0 "$p1 javaCommand start"

$ns_ at 10.0 "finish $ns_"

proc finish {ns_} {
$ns_ halt
delete $ns_
}

$ns_ run
```

The Java classes are located and instantiate as described in Sect. 5.5. Now, we can use the *javaCommand init* instruction to initialize the Java nodes, set the send to address (the node where the data will be sent to) and start the node off, which results in it sending the data.

6.3.2 The Java Side

On the java side *PAICommands.java* implements various instructions to help send data and to trigger timers etc:

```
package pai.examples.ns2;

import pai.broker.CommandInterface;
import pai.broker.PAIAccessInterface;
import pai.api.PAIInterface;
import pai.net.DatagramSocket;
import pai.net.DatagramPacket;
import pai.net.InetAddress;
import pai.impl.PAITimer;
import pai.impl.Logging;
import pai.event.PAISocketEvent;
```

```
import pai.event.PAISocketListener;
import java.net.SocketException;
import java.io.IOException;

public class PAICommands implements CommandInterface, PAIAccessInterface,
                                        PAISocketListener {
    PAIInterface pai;
    String sendTo;
    DatagramSocket s;
    PAITimer t;
    int count=0;

    public void init() {
        try {
            s = pai.addSocket(5555);
            pai.addPAISocketListener(s,this);
        } catch (SocketException e) {
            System.out.println("Error opening socket");
        }
        catch (IOException ep) {
            System.out.println("Error opening socket");
        }
    }

    void start() {
    timerTriggered();  // transmit first packet right away
        }

    public void dataReceived(PAISocketEvent sv) {
        try {
            ++count;
            byte b[] = new byte[15];
            DatagramPacket p = new DatagramPacket(b,b.length);
            pai.receive(s, p);
            if (Logging.isEnabled()) {
                System.out.println("PAICommands: Received" +
                 " PACKET NUMBER ----------------> " + count);
                System.out.println("PAICommands: Received "
                 + new String(p.getData()) +
                         " from " + p.getAddress().getHostAddress());
            }
        } catch (IOException ep) {
            System.out.println("PAICommands: Error opening socket");
        }
    }

    public void timerTriggered() {
        try {
                byte b[] = (new String("Hello Proteus " +
```

```
                String.valueOf(count)).getBytes());
               DatagramPacket p =new DatagramPacket(b, b.length,
                new InetAddress(sendTo), 5555);
               pai.send(s,p);
        }      catch (IOException eh) {
            System.out.println("Error Sending Data");
        }
    }


    public String command(String command, String args[]) {
        if (command.equals("init")) {
            init();
            return "OK";
        }
        else if (command.equals("setSendTo")) {
            sendTo = args[0];
            return "OK";
         }
        else if (command.equals("start")) {
            start();
            return "OK";
        }
       else if (command.equals("trigger")) {
             timerTriggered();
             return "OK";
         }
       else if (command.equals("cleanUp")) {
             pai.cleanUp();
             return "OK";
         }

        return "ERROR";
    }

    public void setPAI(PAIInterface pai) {
        this.pai=pai;
    }
}
```

Firstly, you'll notice that *PAICommands* implements three interfaces:

- **CommandInterface:** so that it understands how to execute commands, as described in the previous chapter.
- **PAIAccessInterface:** (see pai.broker.PAIAccessInterface) this interface is a tagging mechanism that tells the subsystem that your Java object wishes to use the JNI interface. Without this, your object cannot use the efficient memory allocation that the subsystem provides for

managing all Java objects. You could in principle access PAI directly but you'd have to manage pointers yourselves, which would be tedious. Using this interface, the *JavaBroker* notifies you of the instance of the *pai* interface by calling the implemented method from this interface, called **setPAI(PAIInterface pai)**, as illustrated. This allows you to store the pai reference locally and use it within your Java object.

- **PAISocketListener:** this allows your class to be notified when data arrives at a *PAISocket*. Briefly, within Java, you attach yourself (or attach others) as a listener on an object and this results in the notification of certain events when they arrive. To make the semantics clear, you have to implement an interface which enables the source object to notify you when its state changes. This is achieved generally by a listener interface, which PAISocketListener implements. Java listeners are an implementation of a callback mechanism. Within C++ you have to point to actual functions, which Java you attach listeners. The interface looks like this:

```
package pai.event;

public interface PAISocketListener {
    public void dataReceived(PAISocketEvent event);
}
```

which contains one method, *dataReceived* that gets invoked when data arrives at the socket. The *dataReceived* method passes a PAISocket event, which contains details about the socket that issued the event (i.e. you may be a listener to several sockets). Once this event is received, you can use *pai* to retrieve the data, using the *receive* method - which takes the socket as a parameter and a DatagramPacket, which is a container to hold the incoming data (this is the standard Java mechanism for doing this).

Briefly, the object is initialized by creating a socket on port 5555. We then add ourselves as a listener for events from this socket. The *start* method gets invoked when a *start* command is received from the NS2 TCL script, this simply invokes the trigger function, which results in a data packets being sent to the the *sendTo* NS2 node. The *sendTo* variable is set using the *setSendTo* TCL command as described previously.

Within the *dataReceived* method, messages are printing our if logging is enabled. There is a static class in *pai.impl.Logging*, which is set globally for all classes within the JVM to turn on or off comments. If it is enabled then you get a verbose output - the default is that it is set to *on*.

6.4 Example 2: Using the Trigger Mechansim

This is a Java example, which implements the *ProtoApp* scenario, the demonstration class for Protolib. Briefly, a trigger is set off once a second to tell the Java object to send data to another node. When the data is received by the receiving NS-2 node, another Java method is triggered allowing it to read the data using the *PAISocketListener* interface.

The actual trigger mechanism is implemented in C++ but this then triggers a method in the Java object to tell it to read the data. This example also uses the *PAICommands* class. When the C++ trigger times out, it sends a 'trigger' command to the Java object, which results in the timerTriggered() method being called. This is equivalent functionality to ProtoApp, but in Java. However, the actual interface to the timer is set within the NS2 TCL script and not the C++ class, enabling the programmer to change the timer's parameters without having to recompile the whole of NS2.

6.4.1 The TCL Side

The following is the TCL script part of the implementation, which creates two *JavaAgent* NS2 nodes attaches the PAICommands Java object to them, initializes them and then sets up the node that will receive the data by invoking the *setSendTo* command on the first node - node 0 sends data to node 1 in this example. We then start a timer by using

```
$ns_ at 0.0 "$p1 startTimer 1 -1"
```

which sets off a timer that times out once per second and runs forever (i.e. -1 flag). The timer is stopped at the end of the simultation. Here is the TCL script:

```
# Create simulator instance
set ns_ [new Simulator]

# Create two nodes
set n1 [$ns_ node]
set n2 [$ns_ node]

# Put a link between them
$ns_ duplex-link $n1 $n2 64kb 100ms DropTail
$ns_ queue-limit $n1 $n2 100
$ns_ duplex-link-op $n1 $n2 queuePos 0.5
$ns_ duplex-link-op $n1 $n2 orient right
```

```
puts "Creating PAI Broker Agents ..."
# Create two Protean example agents and attach to nodes
set p1 [new Agent/JavaAgent]
$ns_ attach-agent $n1 $p1

set p2 [new Agent/JavaAgent]
$ns_ attach-agent $n2 $p2

puts "CREATED OK          ....... ..."

# Initialize each broker telling it what its NS2 address is

puts "In script: Initializing  ..."

$ns_ at 0.0 "$p1 initAgent"
$ns_ at 0.0 "$p2 initAgent"

$ns_ at 0.0 "$p1 setClass
/Users/scmijt/Apps/nrl/p2ps-ns2/classes pai.examples.ns2.PAICommands"
$ns_ at 0.0 "$p2 setClass
/Users/scmijt/Apps/nrl/p2ps-ns2/classes pai.examples.ns2.PAICommands"

puts "Starting simulation ..."

$ns_ at 0.0 "$p1 javaCommand init"
$ns_ at 0.0 "$p2 javaCommand init"

$ns_ at 0.0 "$p1 javaCommand setSendTo [$n2 node-addr]"

$ns_ at 0.0 "$p1 javaCommand start"

# The timer is started within C++ code NOT Java but the
# parameters are specified here

$ns_ at 0.0 "$p1 startTimer 1 -1"

# Stop
$ns_ at 9.0 "$p1 stopTimer"
$ns_ at 9.0 "$p2 stopTimer"

#Clean up objects

$ns_ at 10.0 "$p1 cleanUp"
$ns_ at 10.0 "$p2 cleanUp"

$ns_ at 10.0 "finish $ns_"

proc finish {ns_} {
```

```
$ns_ halt
delete $ns_
}

$ns_ run
```

This example, will run the timer once per second (well, NS2 second anyway - which is non real-time so in effect one second will be microseconds ....) and iterate for 10 iterations as specified by the NS2 time-stepping as shown.

6.5 Example 3: Sending Data Using Multicast

A Java example, which also implements the ProtoApp scenario but this timer uses a multicast address to send the data between the nodes. The first node sends the data to the multicast address and the second node listens to this address and gets notified when something happens. This example uses the *pai.examples.ns2.MulticastTimerDemo* Java class to implement the Java functionality.

6.5.1 The TCL Side

The following is the TCL script part of the implementation, which creates a muticast enabled NS2 and creates a multicast address for communication. The multicast address to be used must be specified in NS2 and then passed to the Java objects so they know which address to use i.e. by using the *setGroupAddress* java TCL script command as illustrated below. Two *JavaAgent* NS2 nodes are created and attach a *MulticastTimerDemo* object:

```
# Create multicast enabled simulator instance
set ns_ [new Simulator -multicast on]
$ns_ multicast

# Create two nodes
set n1 [$ns_ node]
set n2 [$ns_ node]

# Put a link between them
$ns_ duplex-link $n1 $n2 64kb 100ms DropTail
$ns_ queue-limit $n1 $n2 100
$ns_ duplex-link-op $n1 $n2 queuePos 0.5
$ns_ duplex-link-op $n1 $n2 orient right
```

```
# Configure multicast routing for topology
set mproto DM
set mrthandle [$ns_ mrtproto $mproto  {}]
 if {$mrthandle != ""} {
     $mrthandle set_c_rp [list $n1]
}

# 5) Allocate a multicast address to use
set group [Node allocaddr]

puts "Creating Java Broker Agents ..."
# Create two Protean example agents and attach to nodes
set p1 [new Agent/JavaAgent]
$ns_ attach-agent $n1 $p1

set p2 [new Agent/JavaAgent]
$ns_ attach-agent $n2 $p2

puts "CREATED OK          ....... ..."

# Initialize C++ agents

puts "In script: Initializing  ..."

$ns_ at 0.0 "$p1 initAgent"
$ns_ at 0.0 "$p2 initAgent"

#set up the class

$ns_ at 0.0 "$p1 setClass
/Users/scmijt/Apps/nrl/p2ps-ns2/classes pai.examples.ns2.MulticastTimerDemo"
$ns_ at 0.0 "$p2 setClass
/Users/scmijt/Apps/nrl/p2ps-ns2/classes pai.examples.ns2.MulticastTimerDemo"

puts "Starting simulation ..."

$ns_ at 0.0 "$p1 javaCommand setGroupAddress $group"

$ns_ at 0.0 "$p1 javaCommand init"
$ns_ at 0.0 "$p2 javaCommand init"

$ns_ at 0.0 "$p1 javaCommand start"

# The timer is started within C++ code NOT Java but the
# parameters are specified here

$ns_ at 0.0 "$p1 startTimer 1 -1"

# Stop
```

```
$ns_ at 9.0 "$p1 stopTimer"
$ns_ at 9.0 "$p2 stopTimer"

#Clean up objects

$ns_ at 10.0 "$p1 cleanUp"
$ns_ at 10.0 "$p2 cleanUp"

$ns_ at 10.0 "finish $ns_"

proc finish {ns_} {
$ns_ halt
delete $ns_
}

$ns_ run
```

We then initialize the two *JavaAgent* NS2 nodes start the first node. This results in the first node sending a data packet to the chosen Multicast address, which results in the second node receiving notification of this transfer. The timer is then kicked off, which repeats this process 10 times

6.5.2 The Java Side

On the java side *MulticastTimerDemo.java* implements various commands, rather similar to the PAICommands class, except that it replaces the *setSender* function with the Multicast address, enabling all nodes to talk to a central address. This enables nodes to automatically send data to collections of nodes and it is this process that will enable P2PS to discover the address of other nodes using its discovery mechanisms. The code looks like this:

```
package pai.examples.ns2;

import pai.broker.CommandInterface;
import pai.broker.PAIAccessInterface;
import pai.broker.JavaBroker;
import pai.api.PAIInterface;
import pai.net.DatagramSocket;
import pai.net.DatagramPacket;
import pai.net.InetAddress;
import pai.net.MulticastSocket;
import pai.impl.PAITimer;
import pai.impl.Logging;
import pai.event.PAISocketEvent;
import pai.event.PAISocketListener;
```

```
import java.net.SocketException;
import java.io.IOException;

/**
 * @author Ian Taylor.
 * A demo of a NS2 Java Object that
 */
public class MulticastTimerDemo implements CommandInterface, PAIAccessInterface,
                                        PAISocketListener {
    PAIInterface pai;
    MulticastSocket s;
    PAITimer t;
    int count=0;

    public void init() {

        try {
            s = new MulticastSocket(5555);
            pai.addPAISocketListener(s,this);
            pai.joinGroup(s,
             new InetAddress(JavaBroker.getMulticastAddress()));
        } catch (SocketException e) {
            System.out.println("Error opening socket");
        }
        catch (IOException ep) {
            System.out.println("Error opening socket");
        }
    }

    void start() {
        timerTriggered();  // transmit first packet right away
        }

    public void dataReceived(PAISocketEvent sv) {
        try {
            System.out.println("Receiving ----------------------------");
            ++count;
            byte b[] = new byte[15];
            DatagramPacket p = new DatagramPacket(b,b.length);
            pai.receive(s, p);
            if (Logging.isEnabled()) {
                System.out.println("PAICommands: Received " +
                 "PACKET NUMBER ----------------> " + count);

                System.out.println("PAICommands: Received "
                 + new String(p.getData()) +
                         " from " + p.getAddress().getHostAddress());
            }
        } catch (IOException ep) {
```

```
            System.out.println("PAICommands: Error opening socket");
        }
    }

    public void timerTriggered() {
        try {
            byte b[] = (new String("Hello Proteus " +
                 String.valueOf(count)).getBytes());
            System.out.println("Address is " +
                 JavaBroker.getMulticastAddress());
            DatagramPacket p =new DatagramPacket(b, b.length,
                 new InetAddress(JavaBroker.getMulticastAddress()), 5555);
            pai.send(s,p);
        }      catch (IOException eh) {
            System.out.println("Error Sending Data");
        }
    }

    public String command(String command, String args[]) {
        if (command.equals("init")) {
            init();
            return "OK";
        }
        else if (command.equals("start")) {
            start();
            return "OK";
        }
       else if (command.equals("trigger")) {
             timerTriggered();
             return "OK";
         }
       else if (command.equals("cleanUp")) {
             pai.cleanUp();
             return "OK";
         }
        return "ERROR";
    }

    public void setPAI(PAIInterface pai) {
        this.pai=pai;
    }
}
```

The first thing to notice here is that we are not using the *PAI* Java interface to create our Multicast socket, but we are using the *MulticastSocket* class. The *MulticastSocket* class we are using here is the PAI re-implementation of the java.io.MulticastSocket class for use with our Java PAI interface. The actual implementation of *MulticastSocket* simply calls the PAI interface in

order to create the appropriate socket, that is in this case, it creates a normal DatagramSocket by using the default constructor and sets Multicast to *true* on this socket so that it can join the multicast group address.

The actual multicast group address being used is set from the TCL script, as described. Java NS2 object gain access to this address by using the:

```
JavaBroker.getMulticastAddress();
```

static method call. This enables any Java object within this JVM to gain access to the default Multicast address that it should use. P2PS uses this same address also when communicating with other P2PS nodes, as we will see in Chapt. **??**. Here therefore, we join the Multicast group by issuing the following PAI command:

```
pai.joinGroup(s, new InetAddress(JavaBroker.getMulticastAddress()));
```

The rest of the class simply implements the same functionality as the PAICommands class discuss earlier in this chapter.

6.6 Conclusion

In this chapter, the Java PAI interface was discussed. An overview of the architecture was given and a brief description of how the classes implement this functionality. We then outlined three examples, which show how one would send data between NS2 nodes, how one would use the timing interface to send repeated calls and how one would use a Multicast address to send data to any nodes that are listening to this address.

Chapter 7

P2PS (Peer-to-Peer Simplified)

This chapter was written by Ian Wang (ian@wangy.co.uk) and gives an overview of the P2PS middleware.

7.1 Introduction

P2PS (Peer-to-Peer Simplified) is a lightweight peer-to-peer infrastructure. As its name suggests, P2PS aims to provide a simple infrastructure on which to develop peer-to-peer style applications, hiding the complexity of other similar architectures such as JXTA [4] and JINI [10].

As the P2PS infrastructure is based on XML discovery and communication, it is independent of any implementation language and computing hardware. Assuming that suitable P2PS implementations exist, it should be possible to form a peer network that includes everything from supercomputer peers to PDAs. Furthermore, communication within P2PS is not tied to any single transport protocol, such as TCP/IP, and can be extended to include new protocols, such as Bluetooth. That said, the current reference implementation of P2PS is written in Java and includes endpoint resolvers for the TCP and UDP.

Although P2PS is not an implementation of the JXTA protocols, its architecture is inspired by that of JXTA. However, P2PS focuses only on the core elements required for peer discovery and pipe-based communication, and hopefully avoids the complexity problems some users have experienced with JXTA (in fact, the reason P2PS was developed in the first place was due to problems experienced using JXTA).

In Section 7.2 of this the paper we outline the P2PS infrastructure and the extensible XML advertisements and queries it employs. In Section 7.3 we go on to describe the Java reference implementation of this infrastructure,

however we note that this is only an example and that other P2PS implementations may employ different architectures. We summarize the content of this paper in Section 7.4.

More information on P2PS and downloads of the Java reference implementation can be obtained from:

`http://www.trianacode.org`

7.2 P2PS Architecture

At an abstract level, a P2PS network can simply be seen as a set of peer implementations connected together by endpoint-to-endpoint communication channels. It is not necessary that all the peers in the network use the same P2PS implementation or run on the same platform. It is also not necessary that all the communication channels use the same messaging protocol. The only requirement for a peer to set-up a communication channel is that it knows the endpoint address for the channel on the receiving peer and the protocol expected by the receiving endpoint.

The abstract network described above is not especially useful or dynamic, so P2PS defines a set of extensible advertisements that allow peers to describe themselves and their endpoints to other peers, and a set of extensible queries for querying advertisements published by other peers. We outline the standard advertisement and query types in Sections 7.2.1 and 7.2.2, and outline how peers handle queries in Section 7.2.5.

P2PS also defines a standard mechanism for publishing and discovering advertisements based on discovery pipes and rendezvous peers; we describe this in Sections 7.2.3 and 7.2.4. Finally, we discuss endpoints and how pipe advertisements are resolved into endpoint-to-endpoint communication channels in Sections 7.2.6 and 7.2.7.

7.2.1 Advertisements

A P2PS advertisement is an XML document with the root name denoting the type of advertisement (e.g. PipeAdvertisement). In addition to its type, every advertisement contains two default XML elements:

```
<?xml version="1.0" encoding="UTF-8"?>
<AdvertisementType>
    <advertId>a unique id for the advert</advertId>
    <peerId>the unique id for the peer that issued the
        advert</peerId>
```

```
</AdvertisementType>
```

Additional elements are defined for different advertisement types. The standard P2PS advertisement types are:

Pipe Advertisement - A pipe is a named virtual communication channel that is only bound to specific endpoints at connection time (using an endpoint resolver, see Section 7.2.7). While multiple pipes can share the same name, every pipe is also identified by a unique id. A pipe also has a specified type, the basic pipe types are:

standard - Assumed to be unidirectional, unreliable and not secure; the actual properties of the pipe depend on the transport protocol that it is bound over.

bidirectional - The same as a standard pipe but with bidirectional communication.

discovery - Allows adverts/queries to be broadcast to and received from other discovery pipes within a certain subnet (referred to as the discovery subnet). A typical implementation of a discovery pipe would be UDP multicast, in which case the scope of the discovery subnet would be determined by the local multicast range. We discuss discovery pipes further in Section 7.2.1.

Obvious extensions to the standard pipe types include *reliable* and *secure* pipes.

Service Advertisement - A service is simply a named collection of pipes. As with pipes, while multiple services can share the same name, every service is also identified by a unique id. P2PS services should not be confused with the discovery service, rendezvous service and pipe service described in Sections 7.3; the latter are elements of the Java reference implementation rather than part of the P2PS architecture.

Endpoint Advertisement - An endpoint is an address for sending and receiving data using a specified endpoint protocol.

Endpoint Resolver Advertisement - An endpoint resolver is itself an endpoint that when queried (with an Endpoint Query) returns the endpoint address for a pipe. An endpoint resolver can answer endpoint queries for more than one peer and for more than one protocol. We look at endpoint resolution further in Section 7.2.7.

Rendezvous Advertisement - A rendezvous is a specialized peer that provides query forwarding endpoints. Other rendezvous peers (usually from separate discovery subnets) forward the queries that they receive to these endpoints. This mechanism allows queries to be propagated across multiple discovery subnets. The basic protocol is that only queries, not basic advertisements, are forwarded by rendezvous peers; we discuss this further in Section 7.2.3,

The XML for all the above adverts can be extended to include additional information if required. Also, if required, completely new advertisement types can be created and advertised/discovered. In Appendix 7.6 we outline the XML for all the standard advertisement types.

7.2.2 Queries

Queries are specialized advertisements that are used to locate other advertisements that match the query parameters. For example, a peer wishing to discover all pipes named 'serverPipe' would publish a Pipe Query with the *queryPipeName* element set to 'serverPipe'.

In addition to the standard advertisement XML, a query also includes the following elements:

query - tags the advertisement as a query and specifies which type of advertisements this query is interested in (e.g. 'PipeAdvertisement'). If the query is interested in multiple advertisement types then multiple query elements should be used.

replyPipeAdvertisement - an optional element specifying the pipe that replies to this advertisement should be sent to.

replyEndpointAddress - an optional element specifying the endpoint address that replies to this advertisement should be sent to.

It is expected that a query should include either a reply pipe advertisement or a reply endpoint address to which any advertisements that 'match' the query should be sent. This return pipe/endpoint is generally located on the issuing peer, but this is not a requirement. As queries are themselves advertisements they can be broadcast in the same manor is standard advertisements (see Section 7.2.3).

There are four standard P2PS queries for retrieving advertisements, these are:

Pipe Query - Query for Pipe Advertisements by pipe name, pipe ID and peer ID.

Service Query - Query for Service Advertisements by service name and peer ID.

Endpoint Query - Query for Endpoint Advertisements by pipe ID and endpoint protocol.

Endpoint Resolver Query - Query for Endpoint Resolver Advertisements by pipe type, peer ID and endpoint protocol.

As with advertisements, the standard queries can be extended with additional tags or new completely new query types developed. Whether extending a standard query or creating a new query type the P2PS query naming conventions must be followed as this is used by Discovery Services to match advertisements with queries. The P2PS query naming conventions are:

- An element in the query named *queryPipeName* will match elements named *pipeName* or *PipeName* in the advertisement. If there are multiple *queryPipeName* elements in the query, or multiple *pipeName* elements in the advertisement then there only need be a single match between the sets.
- If the are multiple query elements, for example *queryPipeName* and *queryPipeID*, then there must be a match in the advertisement for each query element.
- All matching is case sensitive.

We look at how all advertisements (including queries) are published and discovered in Sections 7.2.3 and 7.2.4, and then look at how queries are handled by peers in Section 7.2.5. In Appendix 7.7 we outline the XML for all the standard query types.

7.2.3 Advertisement and Discovery

There are no restrictions on how the advertisements outlined in the previous sections are passed between P2PS peers. It would therefore be perfectly acceptable for an advertisement to be e-mailed between peers or to be passed on a floppy disc. However, P2PS does define a standard discovery mechanism based on the discovery pipe type mentioned in Section 7.2.1.

A discovery pipe is a special pipe that allows a peer to broadcast advertisements to other peers within a certain subnet (referred to as the discovery subnet), and also to receive advertisements broadcast by other peers within this subnet. It is up to individual endpoint protocols how this discovery pipe concept is implemented, and the implementation determines the discovery subnet scope. For example, in the UDP endpoint protocol outlined in Section 7.3.5, the discovery pipe endpoints are multicast sockets joined to the same group (232.2.2.2), and therefore all the advertisement sent using a discovery pipe are received by all other discovery endpoints within the multicast range. In this example the scope of the discovery subnet coincides with the UDP multicast range. Note that it is not required that an endpoint protocol provide a discovery pipe implementation.

Although the peers in a discovery subnet can discover each others advertisements, a mechanism is also required to link separate discovery subnets. In P2PS this linking is achieved using specialized rendezvous, as we shall discuss in the next section.

7.2.4 Rendezvous Peers

P2PS defines a mechanism for linking discovery subnets based on queries being forwarded between special rendezvous peers. Each rendezvous peer provides at least one endpoint (see Section 7.2.6) to which other rendezvous peers (usually from other discovery subnets) directly forward queries that they discover. We illustrate several discovery subnets linked through rendezvous peer connections in Figure 7.1.

When a rendezvous peer discovers a new *query* (either via its local discovery subnet or forwarded from another rendezvous peer) it should do three things:

- Forward the query directly to the other rendezvous peers it knows about.
- Store a copy of the query in its local cache.
- Check whether any advertisement in its local cache matches the query, and if so broadcast the query within its local discovery subnet.

When a rendezvous peer discovers a new *advertisement* via its local discovery subnet it should do the following:

- Store a copy of the advertisement in its local cache

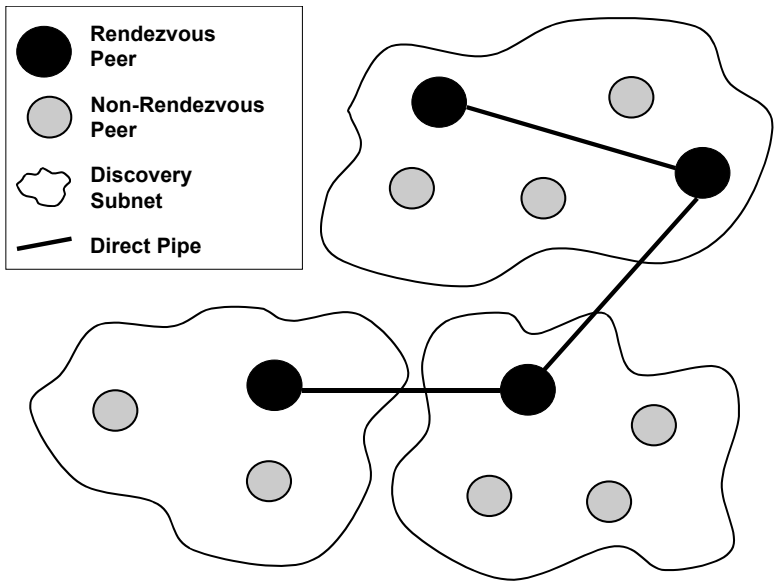

Figure 7.1: Discovery subnets linked through rendezvous peer connections.

- Check whether any query in its local cache matches the advertisement, and if so broadcast that query within its local discovery subnet.

In other words, a rendezvous peer does not answer queries itself; instead it broadcasts the relevant query within its discovery subnet and it is up to the peer that issued the advertisement to provide the answer directly. It is important to note that only queries, not standard advertisements, are forwarded between rendezvous peers. We look at how peers handle queries in the next section.

7.2.5 Query Handling

When a peer discovers a query (i.e. receives a query via its discovery pipe), it is expected to answer that query with any matching advertisements that it have been published locally. So, for example, if a peer receives a query asking for pipe advertisements with the pipe name 'serverPipe' and a pipe advertisement with the pipe name 'serverPipe' was published locally, then that pipe advertisement should be sent as a reply to the query. The P2PS query matching conventions were described in Section 7.2.2.

As mentioned in Section 7.2.2, a query should either include a reply address in the form of a pipe advertisement or an endpoint address. The replying peer should therefore try to connect either an output pipe/output

endpoint to this reply address and send all the matching advertisements via this pipe/endpoint.

7.2.6 Endpoints

In P2PS an endpoint is specified by its endpoint address and the endpoint protocol that it follows. An endpoint is typically a port on a P2PS peer and the endpoint address is the IP address and port number of that port; however other endpoint implementations, such as an email address endpoint, are equally valid.

Both the endpoint address and endpoint protocol are specified as simple strings, with the endpoint protocol indicating the endpoint address format, the message format, and the transport mechanism used. For example, when the protocol is 'UDP' (see Section 7.3.5), the endpoint address takes the form *IP_Addeess:Port_Number* and the messages are sent as datagram packets with an XML header.

A peer can obviously only send to endpoint addresses that use protocols that it recognizes. However, as we shall discuss in the next section, pipes are only bound to specific endpoints at connection time and can operate over multiple protocols.

7.2.7 Pipes and Endpoint Resolution

A pipe is a virtual communication channel that is only bound to specific endpoints at connection time. We refer to the process of determining an endpoint address for a pipe from its pipe id as endpoint resolution.

An endpoint resolver is an endpoint itself that, when sent an endpoint query, returns an endpoint address for the pipe id specified in the query (assuming it knows an endpoint for that pipe). Note the returned endpoint could either be static or dynamically created at query time.

Normally a peer will provide and advertise an endpoint resolver that can resolve endpoints on the pipes advertised by the peer; however this is not essential, and an alternative approach, such as a centralized endpoint resolver that provides endpoint resolution for multiple peers, would be equally valid.

As well as the id of the pipe that an endpoint is required for, an endpoint query also specifies the endpoint protocol the returned endpoint should understand. So, for example, an endpoint query could ask for an 'UDP' endpoint or for a 'TCP' endpoint.

The ids of the peers a resolver provides endpoint addresses for, and the endpoint protocol of those endpoints, are specified in the endpoint resolver

advertisement. It is allowed for multiple resolvers to provide endpoint resolution for the same peer, for instance one resolver could return 'UDP' endpoint addresses and another 'TCP' endpoint addresses. Also allowed would be an endpoint resolver that provides a relay service, i.e. returns the address of an endpoint that simple forwards the messages it receives via another protocol to the actual pipe endpoint. The relay service may itself use an endpoint resolver to determine the actual pipe endpoint.

To clarify, for a peer wishing to connect to the pipe specified in a pipe advertisement, a standard endpoint resolution process would be:

- Find one or more endpoint resolvers for the peer that advertised the pipe. Endpoint resolvers can be discovered by publishing an endpoint resolver query. Note that it is wise for a peer to keep a cache of the endpoint resolver advertisements that it has discovered.
- Send an endpoint query for the required pipe and endpoint protocol to each of the endpoint resolvers.
- If a queried endpoint resolver knows an endpoint for the specified pipe then it will send an endpoint advertisement for the endpoint via the return endpoint/pipe specified in the endpoint query. The connecting peer can then establish the pipe using this endpoint address. Note that it is possible for a peer to receive multiple endpoint replies; in this case it is up to the peer to decide which one to use.

7.3 P2PS Implementation

The P2PS infrastructure is based on XML communication and discovery protocols, and therefore is not tied to any particular implementation language or operating system (see Section 7.2). However, a reference implementation has been developed in Java, and in this section we outline the main elements of this implementation.

We illustrate the overall architecture of the Java reference implementation in Figure 7.2. This implementation is built on five main interfaces:

Peer - Peer is a container interface that allows the application access to the discovery service, rendezvous service and pipe service interfaces, and also to the advertisement factory.

Discovery Service - The discovery service allows the application to publish advertisements/queries. It also allows the application to add discovery listeners that are notified when advertisements/queries are discovered.

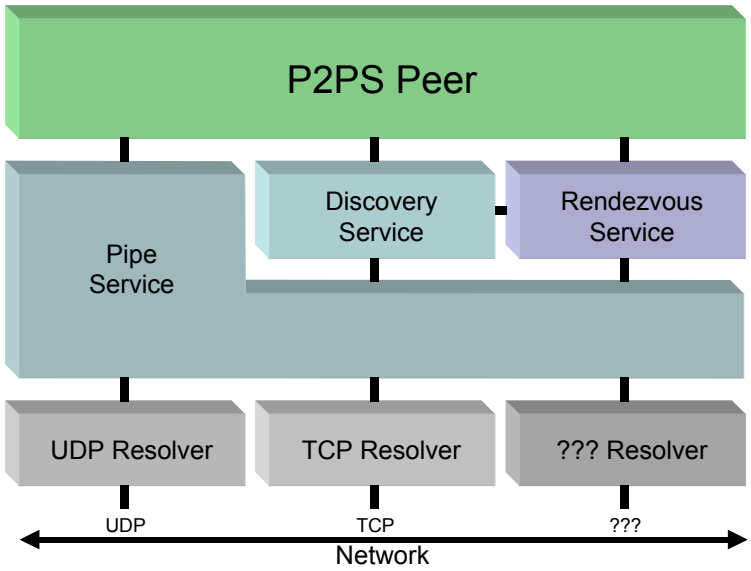

Figure 7.2: The architecture of the P2PS Java reference implementation.

Rendezvous Service - If the peer is running as a rendezvous then the rendezvous service allows the peer to be connected to other rendezvous peers. The rendezvous service handles forwarding queries to/receiving queries from other rendezvous peers.

Pipe Service - The pipe service allows the application (and the discovery and rendezvous services) to create input pipes/connect output pipes. Endpoint resolvers are registered with the pipe service and they perform the actual endpoint creation and resolution.

Endpoint Resolver - Endpoint resolvers create the actual input endpoints for pipes, and also resolve and connect to remote endpoints. There may be multiple endpoint resolvers registered with the pipe service, each creating and resolving endpoints in different endpoint protocols.

In addition to the five above interfaces, there is an Advertisement Factory class for constructing advertisement objects from their XML representation. The discovery service, rendezvous service and pipe service mentioned above are elements of the reference implementation and should not be confused with the P2PS notion of a service discussed in Section 7.2.1.

As the components described above are written as interfaces developers are free to implement their own versions. In particular it is expected that

new implementations of the endpoint resolver interface will be developed to cover additional transport protocols (such as Bluetooth).

In the next sections we briefly describe the key methods contained in the peer (Section 7.3.1), discovery service (Section 7.3.2), rendezvous service (Section 7.3.3), pipe service (Section 7.3.4) and endpoint resolver (Section 7.3.5) interfaces. We discuss peer configuration in Section 7.3.6. In Appendix 7.5 we give the Java code for a simple client/server application example.

7.3.1 Peer

Peer is a simple container interface that allows an application access to the discovery service, rendezvous service and pipe service, and also to the advertisement factory. The methods in peer are:

init() - Initializes the peer by calling the initialization methods on the other components. This method must be called before any of the other methods are called.

getPeerID() - Returns the unique id for the peer.

getAdvertisementFactory() - Returns the advertisement factory.

getDiscoveryService() - Returns an interface to the discovery service.

getRendezvousService() - Returns an interface to the rendezvous service (or null if the peer is not acting as a rendezvous).

getPipeService() - Returns an interface to the pipe service.

The reference peer implementation can be constructed using either a default configuration, a custom configuration, or using a configuration window. We discuss peer configuration further in Section 7.3.6.

7.3.2 Discovery Service

The discovery service allows the application to publish advertisements/queries, and also to add discovery listeners that are notified when advertisements/queries are discovered. It is also the responsibility of the discovery service the answer queries that are received through the discovery pipe, as described in Section 7.2.5.

The methods in the discovery service interface are:

init() - Initializes the discovery service. Discovery service initialization involves the following:

- Create a discovery pipe by calling the createInputPipe() method on the pipe service with a pipe advertisement of type 'discovery'.
- Create an input pipe that is used to receive query replies. This is also done using the createInputPipe() method on the pipe service, but with a pipe advertisement of type 'standard'.
- Publish an advertisement for each of the endpoint resolvers. This is done using the getPipeResolvers() method on the pipe service, and then the getAdvertisement() method on each of the resolvers.

publish(Advertisement advert) - Publishes the advert by sending it using the discovery pipe. If the advertisement is a query and a reply pipe/reply endpoint is not already specified then the reply pipe for the query is set to the discovery services reply pipe.

addDiscoveryListener(DiscoveryListener listener) - Adds a discovery listener to the discovery service. Discovery listeners are notified every time an advertisement/query is received on the discovery pipe.

removeDiscoveryListener(DiscoveryListener listener) - Removes a discovery listener from the discovery service.

7.3.3 Rendezvous Service

If the peer is acting as a rendezvous then the rendezvous service is responsible for maintaining connections with other rendezvous peers and forwarding queries discovered by the discovery service. The rendezvous service provides at least one endpoint so that it can receive queries forwarded by other rendezvous peers

The rendezvous service maintains a cache of all queries it receives, and when an advertisement matching a cached query is discovered by the discovery service the query is republished (using the discovery service). The role of rendezvous peers is discussed in greater detail in Section 7.2.4.

The methods in the rendezvous service interface are:

init() - Initializes the rendezvous service. This involves using the relevant endpoint resolvers to create the rendezvous endpoints, and also using the discovery service to publish a rendezvous advertisement for this service.

connectToRendezvous(EndpointAddress addr) - Connects the rendezvous service to the specified endpoint address (using the relevant endpoint resolver). Once connected an advert for the rendezvous service is forwarded using the endpoint so that a return connection can be established. Also, all the queries cached by the rendezvous service are forwarded to the endpoint.

getLocalRendezvousAddresses() - Returns an array of the local rendezvous endpoints.

getRemoteRendezvousAddresses() - Returns an array of the remote rendezvous endpoints the rendezvous service is connected to.

Due to the close connections between the rendezvous service and the discovery service, in the reference implementation the rendezvous service is an extension of the discovery service.

7.3.4 Pipe Service

The pipe service provides an interface that allows applications to create and connect pipes. However, the pipe service simply maintains a list of endpoint resolvers and the actual endpoint creation and connection is delegated to these resolvers (see Section 7.3.5).

The methods in the pipe service interface are:

init() - Initializes the pipe service.

createInputPipe(PipeAdvertisement pipead) - Creates an input pipe of the type specified in the pipe advertisement. This is done by asking all the endpoint resolvers that handle the pipe type to establish input endpoints for the pipe.

connectOutputPipe(PipeAdvertisement pipead) - Creates an output pipe connected to the advertised pipe. This is done by asking all the endpoint resolvers that handle the pipe type to attempt to resolve and connect an output endpoint to an input endpoints for the pipe. The first output endpoint successfully connected by any resolver is used for the output pipe.

register(EndpointResolver resolver) - Registers an endpoint resolver to create input endpoints and resolve/connect output endpoints.

unregister(EndpointResolver resolver) - Unregisters an endpoint resolver.

getEndpointProtocols() - Returns an array of the endpoint protocols for which there is at least one registered endpoint resolver.

getPipeResolvers() - Returns an array of the registered endpoint resolvers.

getPipeResolver(String protocol) - Returns an endpoint resolver for the specified endpoint protocol (or null if there are none registered).

7.3.5 Endpoint Resolver

Endpoint resolvers are responsible for creating input endpoints and for resolving pipe ids into connected output endpoints (see Section 7.2.7). Each endpoint resolver implementation handles endpoint creation/connection for one or more endpoint protocols; for example, a pipe service could use one endpoint resolver implementation for creating/connecting 'TCP_Protocol' endpoints and a different resolver implementation for creating/connecting 'UDP_Protocol' endpoints.

The methods in the endpoint resolver interface are:

init(Peer peer, Config config) - Initializes the endpoint resolver. This initializes the resolver endpoint to which other endpoint resolvers send their endpoint queries and their replies to endpoint queries issued by the resolver.

createInputEndpoint(String pipeid, String pipetype) - Creates and returns an input endpoint of the specified pipe type and associated with the specified pipe id.

createInputEndpoint(String pipeid, EndpointAddress address) - Creates and returns an endpoint bound to the specified endpoint address, such as when creating an rendezvous endpoint. The protocol and type specified in the endpoint address must be understood by the endpoint resolver or an exception is thrown.

connectOutputEndpoint(String pipeid, EndpointAddress address) - Creates and returns an endpoint connected to the specified endpoint address; the created endpoint is bound to the specified pipe id. The address being connected to must implement a protocol understood by this resolver or an exception is thrown.

resolveEndpoint(String pipeid, EndpointAddress resolveraddr) - Asks the endpoint resolver to resolve an endpoint for the specified

pipe id. This is done by sending an endpoint query for the specified pipe id to the specified resolver address. Any replies to the endpoint query are received through the resolver endpoint and notified to all the endpoint resolution listeners.

addEndpointResolutionListener(EndpointResolutionListener listener) - Adds an endpoint resolution listener. Endpoint resolution listeners are notified each time an endpoint advertisement is received on the resolver endpoint (in reply to an endpoint query issued by the resolver).

removeEndpointResolutionListener(EndpointResolutionListener listener) - Removes an endpoint resolution listener from being notified when a endpoint advertisement reply is received

isInputPipesEnabled(String type) - Returns true if the endpoint resolver can create input endpoints of the specified type.

setInputPipesEnabled(String type, boolean state) Sets whether the endpoint resolver can create input endpoints of the specified type. Note that only types the resolver can handle can be enabled/disabled.

isOutputPipesEnabled(String type) - Returns true if the endpoint resolver can create output endpoints of the specified type.

setOutputPipesEnabled(String type, boolean state) - Sets whether the endpoint resolver can create output endpoints of the specified type. Note that only types the resolver can handle can be enabled/disabled.

getPipeTypes() - Returns an array of all the pipe types the resolver can handle (whether they are enabled or not).

getResolverEndpoint() - Returns the resolver endpoint for the endpoint resolver. The resolver endpoint receives endpoint queries from other resolvers and also replies to endpoint queries issued by the resolver.

getResolverEndpointAddress() - Returns the endpoint address for the resolver endpoint.

getEndpointProtocols() - Returns an array of the endpoint protocols for which the resolver can create/connect endpoints.

getResolverForPeerIDs() - Returns an array of the peers the resolver can resolve endpoint addresses for. In general an endpoint resolver only resolves endpoint addresses for the peer it is connected to, but it is

possible for a resolver to handle resolution for multiple peers, e.g. when the resolver is acting as a relay through a firewall.

getAdvertisement() - Returns an endpoint resolver advertisement for the resolver. The endpoint resolver advertisement specifies the resolver endpoint address, the endpoint protocols handled and the peer ids handled.

In the reference Java implementation there are endpoint resolvers for two protocols: *UDP* and *TCP*. We look at these implementations in Sections 7.3.5 and 7.3.5.

UDP Resolver and Endpoints

The UDP resolver and endpoints implement packet based communication using Javas datagram sockets. In this protocol each endpoint is a separate port; endpoint addresses specified in the form:

```
IP_Address:Port_Number
```

The maximum length of a datagram packet in this protocol is 50000 bytes, with long messages being split over multiple packets. The first *200* bytes of each packet contain an XML header, this takes the form:

```
<?xml version="1.0" encoding="UTF-8"?>
<Header>
    <id>the unique message id</id>
    <len>the total length of the message data</len>
    <pIdx>the packet index </pIdx>
</Header>
```

The rest of the packet bytes contain the message data (or a section of the message data if the data is split over multiple packets).

In addition to standard input and output endpoints, the UDP resolver allows endpoints to be created for discovery pipes. When the pipe type is 'discovery' a multicast socket is instantiated on port 2221 and joined to the multicast group on inet address 232.2.2.2. Note that endpoints for discovery pipes are bidirectional so messages can be sent and received.

TCP Resolver and Endpoints

The TCP resolver and endpoints implement stream based communication using Javas standard sockets. In this protocol multiple endpoints can share a single port, with each endpoint using a port being identified by a unique endpoint ID. Endpoint addresses are specified in the form:

```
IP_Address:Port_Number:Endpoint_ID
```

When a connection to a port is first initialized, an XML message is sent to identify which endpoint this connection represents. The initialization message takes the form:

```
<?xml version="1.0" encoding="UTF-8"?>
<TCPInitMessage>
    <endpointID>endpoint id</endpointID>
</TCPInitMessage>
```

All messages sent using the TCP protocol, including the initialization message, are preceded by an integer (4bytes) that indicate the length of the message (in bytes).

7.3.6 Configuration

In the reference Java implementation, when a peer is first instantiated there are three configuration options: use the default configuration; use the configuration window; or specify a custom configuration. We discuss these three options in the next sections.

Default Configuration

The default peer configuration is the following:

- The peer is not enabled as rendezvous.
- TCP standard input and output pipes are enabled
- UDP discovery pipes are enabled (port 2221)
- The port range is 2223 to 2333

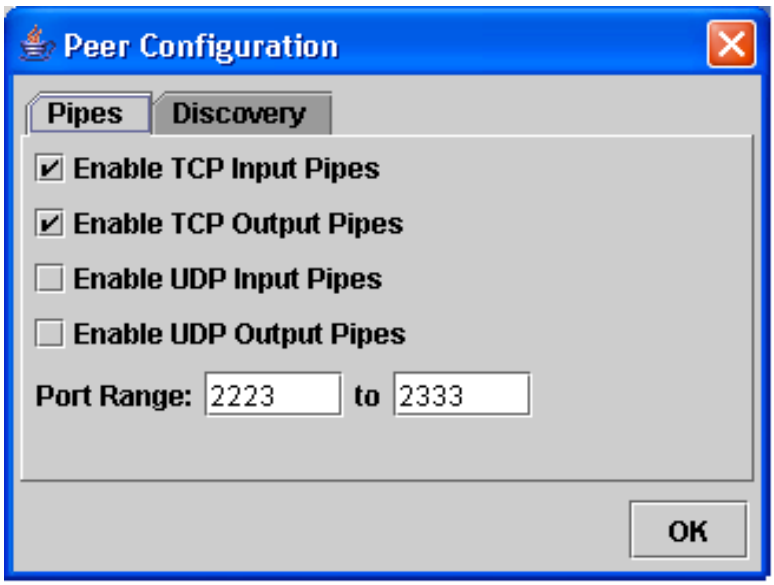

Figure 7.3: The pipe configuration tab.

Configuration Window

An alternative to using the default configuration is to allow the user to configure the peer at instantiation time using the configuration window. The configuration window is split into two tabs, the pipe tab and the discovery tab. The pipe tab (Figure 7.3) allows the user to choose from the following options:

Enable TCP Input Pipes - *(default = enabled)*

Enable TCP Output Pipes - *(default = enabled)*

Enable UDP Input Pipes - *(default = disabled)*

Enable UDP Output Pipes - *(default = disabled)*

Port Range - This specifies the port numbers that will be assigned to endpoints. This option is useful if a firewall is operating and only specific port numbers are open. *(default = 2223 to 2333)*

Although it may seem odd that a user can enable/disable input and output pipes separately, it may be the case that the user wants to create input pipes of only one type but wishes to be able to connect to pipes of multiple types.

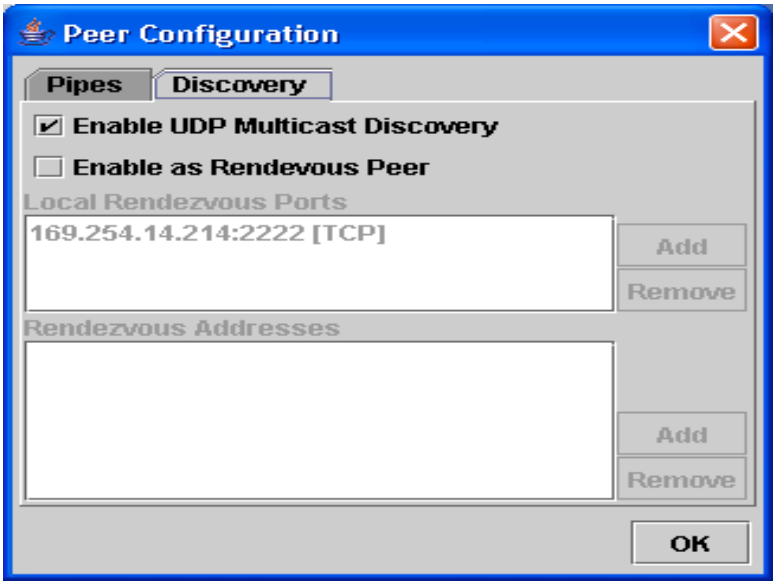

Figure 7.4: The discovery configuration tab.

The discovery tab (Figure 7.4) allows the user to choose from the following options:

Enable UDP Multicast Discovery - This option enables discovery using UDP multicast endpoints (see Section 7.3.5). *(default = enabled)*

Enable as Rendezvous Peer - This option enables the peer as a rendezvous peer (see Section 7.2.4). *(default = disabled)*

Local Rendezvous Ports - If the peer is enabled as a rendezvous then this specifies the ports to which other rendezvous peers forward queries.

Rendezvous Addresses - If the peer is enabled as a rendezvous then this specifies the addresses to which this peer forwards queries.

Custom Configuration

A third option other than using the default configuration or the configuration window is to specify a custom configuration. To use a custom configuration the reference peer is instantiated using an instance on the p2ps.peer.Config interface. The methods contained in the Config interface are:

isRendevousPeer() - Returns true if the peer is a rendezvous.

getLocalRendezvousEndpoints() - Returns an array of the local rendezvous endpoints which other rendezvous peers forward queries to.

getRemoteRendezvousEndpoints() - Returns an array of the remote rendezvous endpoints to which this peer forwards queries.

getResolverConfigs() - Returns an array of resolver configurations. Resolver configurations specify the resolver class name (e.g. p2ps.imp.endpoint.UDP.UDPResolver) and the pipe types that are enabled for that resolver.

getMinPort() - Returns the minimum port in the port range.

getMaxPort() - Returns the maximum port in the port range.

It is obviously up the user how the instance of p2ps.peer.Config is constructed but one possibility is write a custom configuration window and another to use some form of properties file.

7.4 Conclusion

In this paper we have outlined P2PS (Peer-to-Peer Simplified), a lightweight peer-to-peer infrastructure based on XML discovery and communication. The P2PS architecture is inspired by that of Suns JXTA project; however P2PS focuses only on the core elements required for peer discovery and pipe-based communication.

In Section 7.2 we described the P2PS architecture, the standard P2PS advertisement/query types and the mechanisms for publishing and handling these advertisements/queries. We also described the P2PS notion of discovery subnets and rendezvous peers for connecting discovery subnets. In Section 7.3 we looked at the architecture of the Java reference implementation of P2PS and the interfaces it contains. However, we noted that this is only one implementation of the P2PS infrastructure and other implementations are free to employ different architectures.

More information on P2PS and downloads of the Java reference implementation can be obtained from:

`http://www.trianacode.org`

7.5 Client/Server Example

In this section we give an example client/server application built using the Java reference P2PS implementation. In this very simple example the server advertises a pipe called serverPipe, and the client discovers this pipe and sends a simple test message along it to the server.

The Java code for the example server is:

```
public class PeerServer implements MessageListener {

    private AdvertisementFactory adverts;
    private DiscoveryService discovery;
    private PipeService pipes;

    public PeerServer() throws IOException {
        // initialize peer
        Peer peer = new PeerImp(true);
        peer.init();

        // retrieve services
        adverts = peer.getAdvertisementFactory();
        discovery = peer.getDiscoveryService();
        pipes = peer.getPipeService();

        System.out.println("Server Started");

        // initialise server pipe advertisement
        PipeAdvertisement pipead = (PipeAdvertisement)
            adverts.newAdvertisement(PipeAdvertisement.PIPE_ADVERTISEMENT_TYPE);
        pipead.setPipeName("serverPipe");

        // create server pipe and attach listener
        InputPipe inpipe = pipes.createInputPipe(pipead);
        inpipe.addPipeListener(this);

        // publish server pipe advertisement
        discovery.publish(pipead);
    }

    public void messageReceived(MessageReceivedEvent event) {
        // display received messages
        System.out.println("\nMessage: " + new String(event.getMessage()));
```

```
        System.out.println("(Received on " +
            event.getInputPipe().getPipeName() + " " +
            event.getInputPipe().getPipeID() + ")");
    }

    public static void main(String[] args) throws IOException {
        new PeerServer();
    }

}
```

The Java code for the example client is:

```
public class PeerClient implements DiscoveryListener, MessageListener {

    private AdvertisementFactory adverts;
    private DiscoveryService discovery;
    private PipeService pipes;

    public PeerClient() throws IOException {
        // initialize peer
        Peer peer = new PeerImp(true);
        peer.init();

        // retrieve services
        adverts = peer.getAdvertisementFactory();
        discovery = peer.getDiscoveryService();
        pipes = peer.getPipeService();

        // listen for discovered advertisements
        discovery.addDiscoveryListener(this);

        System.out.println("Client Started: Locating Server Pipe");

        // create pipe query for server pipe
        PipeQuery pipequery = (PipeQuery)
            adverts.newAdvertisement(PipeQuery.PIPE_QUERY_TYPE);
        pipequery.setQueryPipeName("serverPipe");

        // publish pipe query
        discovery.publish(pipequery);
    }

    public void advertDiscovered(DiscoveryEvent event) {
```

```
        try {
            Advertisement advert = event.getAdvertisement();

            // handle discovered pipe advertisement
            if (advert instanceof PipeAdvertisement) {
                System.out.println("\nPipe discovered: " +
                    ((PipeAdvertisement) advert).getPipeName() + " " +
                    ((PipeAdvertisement) advert).getPipeID());

                // connect output pipe to discovered pipe
                OutputPipe outpipe =
                    pipes.connectOutputPipe((PipeAdvertisement) advert);

                // send test message
                outpipe.send("HELLO".getBytes());

                System.out.println("Message Sent: HELLO");
            }
        }
        catch(IOException except) {
            except.printStackTrace();
        }
    }

    public void messageReceived(MessageReceivedEvent event) {
        System.out.println("   Message Received on " +
            event.getInputPipe().getPipeName() + " " +
            event.getInputPipe().getPipeID());
    }

    public static void main(String[] args) throws IOException {
        new PeerClient();
    }
}
```

7.6 Advertisements

In this section we give the XML for the standard advertisement types described in Section 7.2.1. The tags shown here are in addition to the <advertId¿ and `<peerId>` tags which are included in every advertisement (see Section 7.2.1).

Pipe Advertisement

```
<?xml version="1.0" encoding="UTF-8"?>
<PipeAdvertisement>
    <pipeId>the pipe id</pipeId>
    <pipeName>the pipe name</pipeName>
    <pipeType>the pipe type (e.g. standard,
        bidirectional, discovery)</pipeType>
</PipeAdvertisement>
```

Service Advertisement

```
<?xml version="1.0" encoding="UTF-8"?>
<ServiceAdvertisement>
    <serviceId>the service id</serviceId>
    <serviceName>the service name</serviceName>
    <controlPipes>
        <PipeAdvertisement>
            advert for a control pipe
        </PipeAdvertisement>
        ... optional additional control adverts
    </controlPipes>
</ServiceAdvertisement>
```

Endpoint Advertisement

```
<?xml version="1.0" encoding="UTF-8"?>
<EndpointAdvertisement>
    <endpointAddress>
        <address>the endpoint address</address>
        <type>the endpoint type (e.g. UNICAST, MULTICAST)</type>
        <protocol>the endpoint protocol expected</protocol>
    </endpointAddress>
    <endpointProtocol>the endpoint protocol expected</endpointProtocol>
    <pipeId>the pipe id this endpoint act for</pipeId>
    ... optional additional endpoint protocols/pipe ids
</EndpointAdvertisement>
```

Endpoint Resolver Advertisement

```
<?xml version="1.0" encoding="UTF-8"?>
<EndpointResolverAdvertisement>
    <endpointAddress>
        <address>the endpoint address for the resolver</address>
        <type>the endpoint type (e.g. UNICAST, MULTICAST)</type>
        <protocol>the endpoint protocol expected</protocol>
    </endpointAddress>
    <resolverForPeerID>the peer endpoints are resolved for</resolverForPeerID>
    <resolverPipeType>the pipe type the resolver handles</resolverPipeType>
    <transportProtocol>the transport protocol the resolver handles</transportProtocol>
    ... optional additional peer ids/pipe types/transport protocols
</EndpointResolverAdvertisement>
```

Rendezvous Advertisement

```
<?xml version="1.0" encoding="UTF-8"?>
<RendezvousAdvertisement>
    <rendezvousAddress>
        <endpointAddress>
            <address>the rendezvous endpoint address</address>
            <type>the endpoint type (e.g. UNICAST, MULTICAST)</type>
            <protocol>the endpoint protocol expected</protocol>
        </endpointAddress>
        ... optional additional rendezvous endpoint addresses
    </rendezvousAddress>
</RendezvousAdvertisement>
```

7.7 Queries

In this section we give the XML for the standard query types described in Section 7.2.2. As noted previously, queries are specialized advertisements, and the tags given here are in addition to the <advertId¿ and `<peerId>` tags which are included in every advertisement (see Section 7.2.1).

Pipe Query

```
<?xml version="1.0" encoding="UTF-8"?>
<PipeQuery>
    <query>PipeAdvertisement</query>
    <queryPipeId>optional pipe id to match</queryPipeId>
    <queryPipeName>optional pipe name to match</queryPipeName>
```

```
    <queryPipeType>optional pipe type to match</queryPipeType>
    ... optional additional query tags
</PipeQuery>
```

Service Query

```
<?xml version="1.0" encoding="UTF-8"?>
<ServiceQuery>
    <query>ServiceAdvertisment</query>
    <queryServiceId>the service id to match</queryServiceId>
    <queryServiceName>the service name to match</queryServiceName>
    ... optional additional query tags
</ServiceQuery>
```

Endpoint Query

```
<?xml version="1.0" encoding="UTF-8"?>
<EndpointQuery>
    <query>EndpointAdverisement</query>
    <queryEndpointProtocol>the endpoint protocol to match</queryEndpointProtocol>
    <queryPipeId>the pipe id to match </queryPipeId>
    ... optional additional query tags
</EndpointQuery>
```

Endpoint Resolver Query

```
<?xml version="1.0" encoding="UTF-8"?>
<EndpointResolverQuery>
    <query>EndpointResolverAdverisement</query>
    <queryResolverForPeerID>the peer id to match </queryResolverForPeerID>
    <queryResolverPipeType>the pipe type to match </queryResolverPipeType>
    <queryTransportProtocol>the transport protocol to match </queryTransportProtoco
    ... optional additional query tags
</EndpointResolverQuery>
```

Chapter 8

Using P2PS within Java Agents

In Chapt. 5, an overview of how Java objects could be attached to Ns2 nodes was given. In the previous Chapter, an overview of the P2PS middleware was given. In this chapter, we show how P2PS can be used from within a Java object that has been attached to an NS2 node for providing higher level P2P discovery and communication services within NS2, through the Java PAI integration outlined in Chapt. 6.

8.1 P2PS Interface to PAI

The P2PS communication layer to the low level sockets and Java Internet packages has been replaced using JNI to invoke the PAI (and hence Protolib) protocols, thereby using the Protolib UDP implementations rather than the native Java ones. An overview of this integration is given in Fig. 6.1.

The customised P2PS version uses the PAI JNI interface in order to interface with Protolib sockets. It also reimplements the Java socket classes in order to provide a unified interface to the underlying sockets. For this implementation, I re-implemented all of the methods in the Java *DatagramPacket*, *DatagramSocket*, *InetAddress* and *MulticastSocket* java.net classes. I simply reimplemented the method calls but put the resulting code within my own *pai.net* package.

Therefore, from a Java programmers perspective, in order to interface with PAI, Protolib and therefore NS2, s/he only has to do a find and replace on all occurrences of the string *java.net* and replace them with *pai.net* and their code will use the Java-PAI version of the socket implementations. Therefore, this infrastructure makes it extremely easy to integrate other middleware systems within this framework. P2PS is an example of showing how things can work but other Java middleware could be easily plugged in at a later stage.

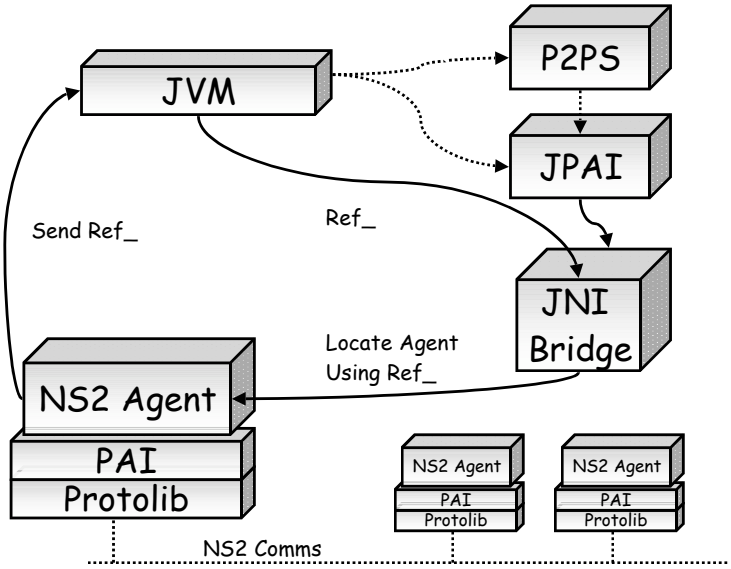

Figure 8.1: An overview of how P2PS is hooked into each node using JNI

8.2 Discovering NS2 Nodes Using P2PS

8.2.1 The TCL Side

The following is the TCL script part of the implementation: which creates two *JavaAgent* NS2 nodes attaches the two Java object to themsand uses the P2PS discovery mechanisms to discover each other and to send a message between the nodes:

```
# Create multicast enabled simulator instance
set ns_ [new Simulator -multicast on]
$ns_ multicast

# Create two nodes
set n1 [$ns_ node]
set n2 [$ns_ node]

# Put a link between them
$ns_ duplex-link $n1 $n2 64kb 100ms DropTail
$ns_ queue-limit $n1 $n2 100
$ns_ duplex-link-op $n1 $n2 queuePos 0.5
$ns_ duplex-link-op $n1 $n2 orient right

# Configure multicast routing for topology
set mproto DM
```

```
set mrthandle [$ns_ mrtproto $mproto  {}]
 if {$mrthandle != ""} {
     $mrthandle set_c_rp [list $n1]
}

# 5) Allocate a multicast address to use
set group [Node allocaddr]

puts "Creating Java Broker Agents and attach them to the NS2 nodes ..."

set p1 [new Agent/JavaAgent]
$ns_ attach-agent $n1 $p1

set p2 [new Agent/JavaAgent]
$ns_ attach-agent $n2 $p2

puts "CREATED OK          ....... ..."

# Initialize C++ agents

puts "In script: Initializing  ..."

$ns_ at 0.0 "$p1 initAgent"
$ns_ at 0.0 "$p2 initAgent"

#set up the Java classes

$ns_ at 0.0 "$p1 setClass
/Users/scmijt/Apps/nrl/p2ps-ns2/classes pai.examples.ns2.P2PSServer"
$ns_ at 0.0 "$p2 setClass
/Users/scmijt/Apps/nrl/p2ps-ns2/classes pai.examples.ns2.P2PSClient"

puts "Starting simulation ..."

$ns_ at 0.0 "$p1 javaCommand setGroupAddress $group"

# Must set Group address BEFORE initializing

$ns_ at 0.0 "$p1 javaCommand init"
$ns_ at 0.0 "$p2 javaCommand init"

# The timer is started within C++ code NOT Java but the
# parameters are specified here

$ns_ at 1.0 "$p1 javaCommand advertise"
$ns_ at 1.0 "$p2 javaCommand query"

$ns_ at 10.0 "$p1 cleanUp"
$ns_ at 10.0 "$p2 cleanUp"
```

```
$ns_ at 10.0 "finish $ns_"

proc finish {ns_} {
$ns_ halt
delete $ns_
}
$ns_ run
```

In this example, we are using two different types of Java nodes, a server and a client. The client uses the P2PS discovery mechanisms to discover the server and then sends it a simple message using the address and port that it has discovered.

Therefore, two different Java classes are located and instantiated, and attached to the two nodes created. We then use the *javaCommand init* instruction to initialize the Java nodes and use then different commands for each node in order to specify its behaviour. The server uses the *javaCommand advertise* instruction to advertise itself on the network, whilst the client uses the *javaCommand query* command in order to publish its query on the network to discover any SRSS nodes available for communication.

8.2.2 The Java Side: The Server

On the java side there are two different Java object being used, a client and a server. This section describes the role of the server, *P2PSServer.java*, which uses P2PS to advertise itself on the network and provide a serving capability via PAI to allow other nodes to communicate with it. The source is listed below:

```
package pai.examples.ns2;

import p2ps.discovery.AdvertisementFactory;
import p2ps.discovery.DiscoveryService;
import p2ps.imp.peer.PeerImp;
import p2ps.peer.Peer;
import p2ps.impl.peer.config.DefaultPAIConfig;
import p2ps.impl.srss.SRSSAdvert;

import java.io.IOException;
import java.net.SocketException;

import pai.broker.CommandInterface;
import pai.broker.PAIAccessInterface;
import pai.api.PAIInterface;
```

```
import pai.impl.Logging;
import pai.event.PAISocketListener;
import pai.event.PAISocketEvent;
import pai.net.DatagramPacket;
import pai.net.DatagramSocket;

public class P2PSServer implements CommandInterface,
        PAIAccessInterface, PAISocketListener {

    private AdvertisementFactory adverts;
    private DiscoveryService discovery;
    PAIInterface pai;
    Peer peer;
    DatagramSocket s;
    int port = 3333;

    public P2PSServer() {
    }

    public void init() {
        Logging.setEnabled(false);

        try {
            peer = new PeerImp(new DefaultPAIConfig());
        } catch (IOException ee) {
            System.out.println("Couldn't create peer");
        }

        try {
            peer.init();
        } catch (IOException ee) {
            System.out.println("Couldn't create peer");
            ee.printStackTrace();
        }

        try {
            s = pai.addSocket(port);
            pai.addPAISocketListener(s, this);
        } catch (SocketException e) {
            System.out.println("Error opening socket");
        } catch (IOException ep) {
            System.out.println("Error opening socket");
        }
    }

    public void advertise() {
        discovery = peer.getDiscoveryService();
        try {
            SRSSAdvert srssadd = new SRSSAdvert("Mobile Node",
```

```
                    pai.getLocalHost().getHostAddress(), port);

            discovery.publish(srssadd);
        } catch (IOException ee) {
            System.out.println("Couldn't create peer");
            ee.printStackTrace();
        }
    }

    public void dataReceived(PAISocketEvent sv) {
        try {
            System.out.println("Receiving ----------------------------");
            byte b[] = new byte[1000];
            DatagramPacket p = new DatagramPacket(b, b.length);
            pai.receive(s, p);
                System.out.println("P2PSServer: Received " +
                        new String(p.getData()) +
                  " from " + p.getAddress().getHostAddress() +
                         " port: " + p.getPort());
        } catch (IOException ep) {
            System.out.println("PAICommands: Error opening socket");
        }
    }

    public String command(String command, String args[]) {
        if (command.equals("init")) {
            init();
            return "OK";
        }
        if (command.equals("advertise")) {
            advertise();
            return "OK";

        } else if (command.equals("cleanUp")) {
            pai.cleanUp();
            return "OK";
        }

        return "ERROR";
    }

    /**
     * Sets the reference to PAI for the direct communication.
     */
    public void setPAI(PAIInterface pai) {
        this.pai = pai;
    }
}
```

The *init()* function first enables the *Logging* to the on position, meaning that detailed comments are output from the supporting Java classes - this is useful for understanding the types of communications which are involved in this example. Although, we are simply advertising ourselves to be discovered by a client, the actual process by which this occurs is pretty complex and involves sending several XML messages between the nodes in order to resolve capabilities etc.

Next, this peer is initialized using a default initialization, which is implemented within the *DefaultPAIConfig()* class, whose implementation is provided below:

```
package p2ps.impl.peer.config;

import p2ps.endpoint.EndpointAddress;
import p2ps.peer.Config;
import p2ps.peer.ResolverConfig;
import p2ps.pipe.PipeTypes;
import p2ps.imp.peer.config.ResolverConfigImp;

import java.io.IOException;

public class DefaultPAIConfig implements Config {

    /**
     * @return true if the peer is a rendezvous peer
     */
    public boolean isRendezvousPeer() {
        return false;
    }

    /**
     * @return an array of the local rendezvous endpoints
     */
    public EndpointAddress[] getLocalRendezvousEndpoints() {
        return new EndpointAddress[0];
    }

    /**
     * @return an array of the remote rendezvous endpoints that the rendezvous
     * service should connect to.
     */
    public EndpointAddress[] getRemoteRendezvousEndpoints() {
        return new EndpointAddress[0];
    }

    /**
     * @return a configuration using a UDP resolver for input, output and
```

```
     * discovery pipes
     */
    public ResolverConfig[] getResolverConfigs() {
        ResolverConfigImp udpconfig =
            new ResolverConfigImp("p2ps.endpoint.NativeUDP.UDPResolver"
                , false, false);

        ResolverConfigImp tcpconfig =
            new ResolverConfigImp("p2ps.endpoint.NativeUDP.UDPResolver"
                , false, false);

        udpconfig.setInputPipesEnabled(PipeTypes.DISCOVERY, true);
        udpconfig.setOutputPipesEnabled(PipeTypes.DISCOVERY, true);

        udpconfig.setInputPipesEnabled(PipeTypes.STANDARD, true);
        udpconfig.setOutputPipesEnabled(PipeTypes.STANDARD, true);

        return new ResolverConfig[] {udpconfig};
    }

    /**
     * @return the minimum available port number
     */
    public int getMinPort() {
        return DEFAULT_MIN_PORT;
    }

    /**
     * Read a saved configuration from a file
     */
    public void readConfig() throws IOException {
        //no-op no point in saving a default
    }

    /**
     * Write a configuration to a file
     */
    public void writeConfig() throws IOException {
        //no-op no point in saving a default
    }

    /**
     * @return the maximum available port number
     */
    public int getMaxPort() {
        return DEFAULT_MAX_PORT;
    }
}
```

The most important method for SRSS application here, are the setting of the rendezvous flag and the setting of the PAI implementation of the USP Endpoint protocol. One could easily duplicate this class or provide a set method in order to manually set a node to act as a rendezvous node.

Notice here we add new resolvers for our endpoints by setting the new P2PS implementation of the UDP endpoint protocol to override the default Java implementation (i.e. p2ps.endpoint.NativeUDP.UDPResolver). In this configuration, we disable all other endpoint implementations (i.e. Java UDP and TCP) so that we only use our new UDP implementation that uses PAI and the underlying Protolib implementation for passing data between NS2 nodes.

We then initialize the P2PS peer, which sets up the communication sockets for discovery etc and starts the various services that are enabled on this peer.

Finally, we then open a UDP socket using PAI directly, so that our client can communicate with this node once it has discovered its address and port number, which it wishes to use for communication.

The *advertise* method then gets a reference to the discovery service ready to publish the advert, advertising this node's capability on the network. The actual advert used in this example uses a customised advert for SRSS nodes, which has been implemented in a class called SRSSAdvert:

```
SRSSAdvert srssadd = new SRSSAdvert("Mobile Node",
        pai.getLocalHost().getHostAddress(), port);
```

An *SRSSAdvert* creates an XML document, which makes the identifier (i.e. a string), the address and port number of this node available via the discovery service on the network. The SRSSAdvert could be easily extended to add other information which you may want to publish, say for example, to publish node which have different roles within the network. The SRSSAdvert class is given below:

```
package p2ps.impl.srss;

import org.jdom.Element;
import java.io.IOException;
import p2ps.discovery.Advertisement;

/**
 * A basic implementation of an SRSS advert, sued to publish
 * an SRSS node and its capabilities on a network.
 */
public class SRSSAdvert implements Advertisement {
```

```
public static String SRSS_ADVERT="SRSS_Mobile_Node";
public static String SRSS_PORT_TYPE ="STANDARD_UDP_PORT";

private String advertid;
private String peerid;
private int port;

/**
 * Creates an SRSS Advert within an advert ID (i.e. type of SRSS advert),
 * a peer ID i.e. its NS2 address and a port on whichi it communicates.
 *
 * @param advertid
 * @param peerID
 * @param port
 */
public SRSSAdvert(String advertid, String peerID, int port) {
    this.advertid = advertid;
    this.peerid = peerID;
    this.port = port;
}

public SRSSAdvert(Element root) {
    Element elem = root.getChild(ADVERT_ID_TAG);
    if (elem != null)
        advertid = elem.getText();

    elem = root.getChild(PEER_ID_TAG);
    if (elem != null)
        peerid = elem.getText();

    elem = root.getChild(SRSS_PORT_TYPE);
    if (elem != null)
        port = Integer.valueOf(elem.getText()).intValue();
}

/**
 * Output the advert as an xml document
 */
public Element getXMLElement() throws IOException {
    Element root = new Element(SRSS_ADVERT);

    Element elem = new Element(ADVERT_ID_TAG);
    elem.addContent(advertid);
    root.addContent(elem);

    elem = new Element(PEER_ID_TAG);
    elem.addContent(peerid);
    root.addContent(elem);
```

```
        elem = new Element(SRSS_PORT_TYPE);
        elem.addContent(String.valueOf(port));
        root.addContent(elem);

        return root;
    }
    /**
     * @return the type for this advertisement
     */
    public String getType() {
        return SRSS_ADVERT;
    }

    /**
     * @return the unique id for this advertisement
     */
    public String getAdvertID() {
        return advertid;
    }

    /**
     * @return the id (i.e. NS2 network address)
     * of the peer that created this advertisement
     */
    public String getPeerID() {
        return peerid;
    }

    /**
     * @return the id of the port
     */
    public String getPortID() {
        return SRSS_PORT_TYPE;
    }

    /**
     * @return the name of the port
     */
    public int getPort() {
        return port;
    }
}
```

The *SRSSAdvert* basically provides a minimal implementation of an SRSS advert, which allows the inclusion of a simple descriptive string and the address and port number of the NS2 node that it can be communicated with. The bulk of this implementation is to allow the serialization and deserial-

ization of this information to and from an XML format. The XML format creates a common text-based format for exchanging message between P2PS processors, thereby enabling a language-independent format for communication. This, in turn, can enable P2PS nodes to be implemented in different programming languages e.g. a Java P2PS node could easily talk to a C++ one. The serialization and deserialization is achieved using the *jdom* packages, which is the only Jar file needed for the P2PS implementation.

Once, this advert is created, it is published using the discovery service's *publish* method, as illustrated in the program listing above.

The *dataReceived* is then called every time data is sent to this server node by using the *PAISocketListener* interface mechanism, described in previous chapters. This method simply reads the data packet and prints it to the screen.

8.2.3 The Java Side: The Client

The *P2PSClient.java* implementation finds the server and sends it a message. The code listing is as follows:

```
package pai.examples.ns2;

import p2ps.discovery.Advertisement;
import p2ps.discovery.AdvertisementFactory;
import p2ps.discovery.DiscoveryEvent;
import p2ps.discovery.DiscoveryListener;
import p2ps.discovery.DiscoveryService;
import p2ps.imp.peer.PeerImp;
import p2ps.peer.Peer;
import p2ps.impl.peer.config.DefaultPAIConfig;
import p2ps.impl.srss.SRSSQuery;
import p2ps.impl.srss.SRSSAdvert;

import java.io.IOException;
import java.net.SocketException;

import pai.broker.CommandInterface;
import pai.broker.PAIAccessInterface;
import pai.api.PAIInterface;
import pai.net.DatagramPacket;
import pai.net.InetAddress;
import pai.net.DatagramSocket;

public class P2PSClient implements CommandInterface, PAIAccessInterface,
        DiscoveryListener {

    private AdvertisementFactory adverts;
```

```
private DiscoveryService discovery;
PAIInterface pai;
DatagramSocket s;
int portToSendTo = -1;
String addressToSendTo = null;

public P2PSClient() {
}

public void init() throws IOException {
    Peer peer = new PeerImp(new DefaultPAIConfig());
    peer.init();

    discovery = peer.getDiscoveryService();
    discovery.addDiscoveryListener(this);

    try {
          s = pai.addSocket(3333); // for sending to
      } catch (SocketException e) {
          System.out.println("Error opening socket");
      } catch (IOException ep) {
          System.out.println("Error opening socket");
      }

}

public void query() throws IOException {
    SRSSQuery query = new SRSSQuery("Mobile Node");
    discovery.publish(query);
}

public void advertDiscovered(DiscoveryEvent event) {
    Advertisement advert = event.getAdvertisement();

    if (advert instanceof SRSSAdvert) {
        System.out.println("P2PSClient: ------------------> Advert discovered: "
                + advert.getPeerID() + " " + ((SRSSAdvert) advert).getPort());
        // set sending address
        portToSendTo = ((SRSSAdvert) advert).getPort();
        addressToSendTo = advert.getPeerID();
        sendData(); // send it a hello message !
    }
}

public void sendData() {
    try {
        byte b[] = (new String("Hello Proteus").getBytes());
        DatagramPacket p = new DatagramPacket(b, b.length,
                        new InetAddress(addressToSendTo), portToSendTo);
```

```
            pai.send(s, p);
        } catch (IOException eh) {
            System.out.println("Error Sending Data");
        }
    }

    public String command(String command, String args[]) {
        if (command.equals("init")) {
            try {
                init();
            } catch (IOException ee) {
                System.out.println("Could not instantiate peer");
                ee.printStackTrace();
            }
            return "OK";
        }
        if (command.equals("query")) {
            try {
                query();
            } catch (IOException ee) {
                System.out.println("Could not instantiate peer");
                ee.printStackTrace();
            }
            return "OK";
        } else if (command.equals("cleanUp")) {
            pai.cleanUp();
            return "OK";
        }

        return "ERROR";
    }

    /**
     * Sets the reference to PAI for the direct communication.
     */
    public void setPAI(PAIInterface pai) {
        this.pai = pai;
    }
}
```

The init() function loads the default configuration for a P2PS peer and initializes the peer's services. It then obtains a reference to the discovery service and adds itself as a discovery listener using the *addDiscoveryListener* method on the discovery service. This informs the discovery service that this object wishes to be notified every time an advert is discovered on the network. When an advert has been discovered the discovery service notifies this object by invoking the *advertDiscovered(DiscoveryEvent event)* method and passing it an event containing the advert.

The init() method then uses the reference to PAI in order to create a UDP socket on port 3333 for sending a message to the server, once its location has been discovered.

The *query* method then issues an SRSS query in order to discover any adverts that match the specified criteria:

```
SRSSQuery query = new SRSSQuery("Mobile Node");
```

The criteria are specified within the *SRSSQuery* class, which is an implementation of the query counterpart for the SRSS advert class. This class basically, implements a matching function that can be used to match compatible adverts when a query is issued. The class is implemented as follows:

```
package p2ps.impl.srss;

import org.jdom.Element;
import java.io.IOException;
import p2ps.pipe.PipeAdvertisement;
import p2ps.endpoint.EndpointAddress;
import p2ps.discovery.Query;

/**
 * An implementation an SRSS Query, which is a query that
 * is used to search for valid SRSSAdverts.
 */
public class SRSSQuery implements Query {
    private String advertid;
    private String peerid;
    private String querypeerid;

    public SRSSQuery(String advertid) {
        this.advertid = advertid;
    }

    public SRSSQuery(Element root) throws IOException {
        Element elem = root.getChild(ADVERT_ID_TAG);
        if (elem != null)
            advertid = elem.getText();

        elem = root.getChild(PEER_ID_TAG);
        if (elem != null)
            peerid = elem.getText();

    }

    /**
     * @return the type for this advertisement
```

```
     */
    public String getType() {
        return SRSSAdvert.SRSS_ADVERT;
    }

    /**
     * @return the unique id for this advertisement
     */
    public String getAdvertID() {
        return advertid;
    }

    /**
     * @return the id of the peer that created this advertisement
     */
    public String getPeerID() {
        return peerid;
    }

    /**
     * @return the id of the peer this query is interested in (null if any)
     */
    public String getQueryPeerID() {
        return querypeerid;
    }

    /**
     * Sets the id of the peer this query is interested in (null if any)
     */
    public void setQueryPeerID(String id) {
        querypeerid = id;
    }

    /**
     * @return optional pipe for the query reply.
     */
    public PipeAdvertisement getReplyPipeAdvertisement() {
        return null;
    }

    /**
     * Ssets the optional endpoint address for the query reply.
     */
    public void setReplyPipeAdvertisement(PipeAdvertisement pipead) {
    }

    /**
     * @return optional endpoint address for the query reply. If not specified
     * the query matches should be (re)published.
```

```
     */
    public EndpointAddress getReplyEndpointAddress() {
        return null;
    }

    /**
     * Ssets the optional endpoint address for the query reply. If not specified
     * the query matches should be (re)published.
     */
    public void setReplyEndpointAddress(EndpointAddress replyaddr) {
    }

    /**
     * @return true if the specified advert matches the query
     */
    public boolean isMatch(p2ps.discovery.Advertisement advert) {
        if (!(advert instanceof SRSSAdvert))
            return false;

        if (((querypeerid != null)) && (!querypeerid.equals(advert.getPeerID())))
            return false;

        return true;
    }

    /**
     * Output the advert as an xml document
     */
    public Element getXMLElement() throws IOException {
        Element root = new Element(SRSSAdvert.SRSS_ADVERT);

        Element elem = new Element(QUERY_TAG);
        elem.addContent(String.valueOf(true));
        root.addContent(elem);

        elem = new Element(ADVERT_ID_TAG);
        elem.addContent(advertid);
        root.addContent(elem);

        elem = new Element(PEER_ID_TAG);
        elem.addContent(peerid);
        root.addContent(elem);

        return root;
    }
}
```

The key method in the *SRSSQuery* class is the *isMatch* method, which is used to match any adverts with this query. In this implementation, I simply

check whether the advert is an *SRSSAdvert* and that the peer that issued the advert is not the same as the one that issued the query, otherwise you would be locating your own adverts, which is almost certainly undesirable.

When an advert is discovered, the actual instance of the advert is obtained from the event. In the *advertDiscovered* method, we therefore simply check that this advert is indeed an *SRSSAdvert*. If it is, we use the access methods to obtain its address and port number, save these within the client's instance variables and invoke the sendData method, which uses PAI and these discovered address and port numbers to send a message directly to the discovered server.

The resulting output should look something like this (depending on whether you have Logging enabled - here I have enabled it):

```
[Ian-Taylors-Computer:examples/pai/javaAgent] scmijt% ns P2PSDiscovery.tcl
Creating Java Broker Agents and attach them to the NS2 nodes ...
PAIEnvironment: CREATING NEW ENVIRONMENT
CREATED OK          ....... ...
In script: Initializing  ...
Starting simulation ...
Classpath is -Djava.class.path=/Users/scmijt/Apps/nrl/p2ps-ns2/classes
JavaBroker.java: Classname: pai.examples.ns2.P2PSServer
JavaBroker.java: Object wants PAI access .. setting up now...
PAINative: Java library path =
.:/Library/Java/Extensions:/System/Library/Java/Extensions:/usr/lib/java
PAINative: constructor
PAIEnvironment: CREATING NEW ENVIRONMENT
JavaBroker.java: Classname: pai.examples.ns2.P2PSClient
JavaBroker.java: Object wants PAI access .. setting up now...
JavaBroker: Group Address set to -2147483648
PAINative: setNs2Node
P2PSServer: ------------------> Server initialising
PAINative: addPAISocketListener
P2PSServer: ------------------> Peer Created
PAINative: joingroup, address is -2147483648
PAINative: addPAISocketListener
PAINative: addPAISocketListener
PAINative: send, Host Address -2147483648
PAINative: send, Port 5555
PAINative: send, Data <?xml version="1.0" encoding="UTF-8"?>
<ServiceAdvertisement>
    <advertId>127.0.0.1-1082546529171-932091874</advertId>
    <peerId>127.0.0.1-1082546529051-721554874</peerId>
    <serviceName>DiscoveryService</serviceName>
    <serviceID>127.0.0.1-1082546529195-218544900</serviceID>
    <controlPipe>
        <PipeAdvertisement>
            <advertId>127.0.0.1-1082546529164-555487317</advertId>
```

```
            <peerId>127.0.0.1-1082546529051-721554874</peerId>
            <pipeID>127.0.0.1-1082546529164-317733729</pipeID>
            <pipeName>DiscoveryService</pipeName>
            <pipeType>standard</pipeType>
        </PipeAdvertisement>
    </controlPipe>
</ServiceAdvertisement>
PAINative: send, Data Length 702
PAINative: send, Host Address -2147483648
PAINative: send, Port 5555
PAINative: send, Data <?xml version="1.0" encoding="UTF-8"?>
<EndpointResolverAdvertisement>
    <advertId>127.0.0.1-1082546529132-956257422</advertId>
    <peerId>127.0.0.1-1082546529051-721554874</peerId>
    <endpointAddress>
        <address>0:2223</address>
        <endpointType>unicast</endpointType>
        <transportProtocol>UDP</transportProtocol>
    </endpointAddress>
    <resolverPipeTypes>standard</resolverPipeTypes>
    <resolverPipeTypes>discovery</resolverPipeTypes>
    <resolverForPeerID>127.0.0.1-1082546529051-721554874</resolverForPeerID>
    <transportProtocol>UDP</transportProtocol>
</EndpointResolverAdvertisement>
PAINative: send, Data Length 634
PAINative: send, Host Address -2147483648
PAINative: send, Port 5555
PAINative: send, Data <?xml version="1.0" encoding="UTF-8"?>
<EndpointResolverQuery>
    <query>true</query>
    <advertId>127.0.0.1-1082546529308-453632564</advertId>
    <peerId>127.0.0.1-1082546529051-721554874</peerId>
    <queryPeerID>127.0.0.1-1082546529051-721554874</queryPeerID>
    <replyEndpointAddress>
        <endpointAddress>
            <address>0:2223</address>
            <endpointType>unicast</endpointType>
            <transportProtocol>UDP</transportProtocol>
        </endpointAddress>
    </replyEndpointAddress>
</EndpointResolverQuery>
PAINative: send, Data Length 553
PAINative: addPAISocketListener
PAINative: send, Host Address 0
PAINative: send, Port 2223
PAINative: send, Data <?xml version="1.0" encoding="UTF-8"?>
<EndpointResolverAdvertisement>
    <advertId>127.0.0.1-1082546529132-956257422</advertId>
    <peerId>127.0.0.1-1082546529051-721554874</peerId>
```

```
    <endpointAddress>
        <address>0:2223</address>
        <endpointType>unicast</endpointType>
        <transportProtocol>UDP</transportProtocol>
    </endpointAddress>
    <resolverPipeTypes>standard</resolverPipeTypes>
    <resolverPipeTypes>discovery</resolverPipeTypes>
    <resolverForPeerID>127.0.0.1-1082546529051-721554874</resolverForPeerID>
    <transportProtocol>UDP</transportProtocol>
</EndpointResolverAdvertisement>
PAINative: send, Data Length 634
PAINative: dataReceivedAtNativeSocket
PAINative: setNs2Node
PAINative: notifying Listeners
PAINative: setDatagramDetails
PAINative: setDatagramDetails: Port - -1 address is 0
PAINative: recv, Host Address 0
PAINative: recv, Port 2225
PAINative: recv, Data <?xml version="1.0" encoding="UTF-8"?>
<EndpointResolverAdvertisement>
    <advertId>127.0.0.1-1082546529132-956257422</advertId>
    <peerId>127.0.0.1-1082546529051-721554874</peerId>
    <endpointAddress>
        <address>0:2223</address>
        <endpointType>unicast</endpointType>
        <transportProtocol>UDP</transportProtocol>
    </endpointAddress>
    <resolverPipeTypes>standard</resolverPipeTypes>
    <resolverPipeTypes>discovery</resolverPipeTypes>
    <resolverForPeerID>127.0.0.1-1082546529051-721554874</resolverForPeerID>
    <transportProtocol>UDP</transportProtocol>
</EndpointResolverAdvertisement>
PAINative: recv, Data Length 634
PAINative: setNs2Node
PAINative: removeSocket
PAINative: removePAISocketListener
PAINative: addSocket on port 3333
PAINative: addPAISocketListener
PAINative: setNs2Node
P2PSClient: ------------------> creating
PAINative: addPAISocketListener
P2PSClient: ------------------> Peer initialising
PAINAtive: joingroup, address is -2147483648
PAINative: addPAISocketListener
PAINative: addPAISocketListener
PAINative: send, Host Address -2147483648
PAINative: send, Port 5555
PAINative: send, Data <?xml version="1.0" encoding="UTF-8"?>
<ServiceAdvertisement>
```

```
    <advertId>127.0.0.1-1082546529556-853727765</advertId>
    <peerId>127.0.0.1-1082546529534-355209911</peerId>
    <serviceName>DiscoveryService</serviceName>
    <serviceID>127.0.0.1-1082546529556-385878417</serviceID>
    <controlPipe>
        <PipeAdvertisement>
            <advertId>127.0.0.1-1082546529551-522719787</advertId>
            <peerId>127.0.0.1-1082546529534-355209911</peerId>
            <pipeID>127.0.0.1-1082546529551-283564741</pipeID>
            <pipeName>DiscoveryService</pipeName>
            <pipeType>standard</pipeType>
        </PipeAdvertisement>
    </controlPipe>
</ServiceAdvertisement>
PAINative: send, Data Length 702
PAINative: send, Host Address -2147483648
PAINative: send, Port 5555
PAINative: send, Data <?xml version="1.0" encoding="UTF-8"?>
<EndpointResolverAdvertisement>
    <advertId>127.0.0.1-1082546529540-214271659</advertId>
    <peerId>127.0.0.1-1082546529534-355209911</peerId>
    <endpointAddress>
        <address>1:2223</address>
        <endpointType>unicast</endpointType>
        <transportProtocol>UDP</transportProtocol>
    </endpointAddress>
    <resolverPipeTypes>standard</resolverPipeTypes>
    <resolverPipeTypes>discovery</resolverPipeTypes>
    <resolverForPeerID>127.0.0.1-1082546529534-355209911</resolverForPeerID>
    <transportProtocol>UDP</transportProtocol>
</EndpointResolverAdvertisement>
PAINative: send, Data Length 634
PAINative: send, Host Address -2147483648
PAINative: send, Port 5555
PAINative: send, Data <?xml version="1.0" encoding="UTF-8"?>
<EndpointResolverQuery>
    <query>true</query>
    <advertId>127.0.0.1-1082546529633-83243233</advertId>
    <peerId>127.0.0.1-1082546529534-355209911</peerId>
    <queryPeerID>127.0.0.1-1082546529534-355209911</queryPeerID>
    <replyEndpointAddress>
        <endpointAddress>
            <address>1:2223</address>
            <endpointType>unicast</endpointType>
            <transportProtocol>UDP</transportProtocol>
        </endpointAddress>
    </replyEndpointAddress>
</EndpointResolverQuery>
PAINative: send, Data Length 552
```

```
PAINative: addPAISocketListener
PAINative: send, Host Address 1
PAINative: send, Port 2223
PAINative: send, Data <?xml version="1.0" encoding="UTF-8"?>
<EndpointResolverAdvertisement>
    <advertId>127.0.0.1-1082546529540-214271659</advertId>
    <peerId>127.0.0.1-1082546529534-355209911</peerId>
    <endpointAddress>
        <address>1:2223</address>
        <endpointType>unicast</endpointType>
        <transportProtocol>UDP</transportProtocol>
    </endpointAddress>
    <resolverPipeTypes>standard</resolverPipeTypes>
    <resolverPipeTypes>discovery</resolverPipeTypes>
    <resolverForPeerID>127.0.0.1-1082546529534-355209911</resolverForPeerID>
    <transportProtocol>UDP</transportProtocol>
</EndpointResolverAdvertisement>
PAINative: send, Data Length 634
PAINative: dataReceivedAtNativeSocket
PAINative: setNs2Node
PAINative: notifying Listeners
PAINative: setDatagramDetails
PAINative: setDatagramDetails: Port - 2225 address is 1
PAINative: recv, Host Address 1
PAINative: recv, Port 2225
PAINative: recv, Data <?xml version="1.0" encoding="UTF-8"?>
<EndpointResolverAdvertisement>
    <advertId>127.0.0.1-1082546529540-214271659</advertId>
    <peerId>127.0.0.1-1082546529534-355209911</peerId>
    <endpointAddress>
        <address>1:2223</address>
        <endpointType>unicast</endpointType>
        <transportProtocol>UDP</transportProtocol>
    </endpointAddress>
    <resolverPipeTypes>standard</resolverPipeTypes>
    <resolverPipeTypes>discovery</resolverPipeTypes>
    <resolverForPeerID>127.0.0.1-1082546529534-355209911</resolverForPeerID>
    <transportProtocol>UDP</transportProtocol>
</EndpointResolverAdvertisement>
PAINative: recv, Data Length 634
PAINative: setNs2Node
PAINative: removeSocket
PAINative: removePAISocketListener
PAINative: addSocket on port 3333
PAINative: dataReceivedAtNativeSocket
PAINative: setNs2Node
PAINative: notifying Listeners
PAINative: setDatagramDetails
PAINative: setDatagramDetails: Port - 2225 address is 0
```

```
PAINative: recv, Host Address 0
PAINative: recv, Port 5555
PAINative: recv, Data <?xml version="1.0" encoding="UTF-8"?>
<ServiceAdvertisement>
    <advertId>127.0.0.1-1082546529171-932091874</advertId>
    <peerId>127.0.0.1-1082546529051-721554874</peerId>
    <serviceName>DiscoveryService</serviceName>
    <serviceID>127.0.0.1-1082546529195-218544900</serviceID>
    <controlPipe>
        <PipeAdvertisement>
            <advertId>127.0.0.1-1082546529164-555487317</advertId>
            <peerId>127.0.0.1-1082546529051-721554874</peerId>
            <pipeID>127.0.0.1-1082546529164-317733729</pipeID>
            <pipeName>DiscoveryService</pipeName>
            <pipeType>standard</pipeType>
        </PipeAdvertisement>
    </controlPipe>
</ServiceAdvertisement>
PAINative: recv, Data Length 702
P2PSClient: Discovered a ServiceAdvertisement
P2PSClient: Advert class is a p2ps.imp.service.ServiceAdvertisementImp
PAINative: setNs2Node
PAINative: dataReceivedAtNativeSocket
PAINative: setNs2Node
PAINative: notifying Listeners
PAINative: setDatagramDetails
PAINative: setDatagramDetails: Port - 5555 address is 1
PAINative: recv, Host Address 1
PAINative: recv, Port 5555
PAINative: recv, Data <?xml version="1.0" encoding="UTF-8"?>
<ServiceAdvertisement>
    <advertId>127.0.0.1-1082546529556-853727765</advertId>
    <peerId>127.0.0.1-1082546529534-355209911</peerId>
    <serviceName>DiscoveryService</serviceName>
    <serviceID>127.0.0.1-1082546529556-385878417</serviceID>
    <controlPipe>
        <PipeAdvertisement>
            <advertId>127.0.0.1-1082546529551-522719787</advertId>
            <peerId>127.0.0.1-1082546529534-355209911</peerId>
            <pipeID>127.0.0.1-1082546529551-283564741</pipeID>
            <pipeName>DiscoveryService</pipeName>
            <pipeType>standard</pipeType>
        </PipeAdvertisement>
    </controlPipe>
</ServiceAdvertisement>
PAINative: recv, Data Length 702
PAINative: setNs2Node
PAINative: dataReceivedAtNativeSocket
PAINative: setNs2Node
```

```
PAINative: notifying Listeners
PAINative: setDatagramDetails
PAINative: setDatagramDetails: Port - 5555 address is 0
PAINative: recv, Host Address 0
PAINative: recv, Port 5555
PAINative: recv, Data <?xml version="1.0" encoding="UTF-8"?>
<EndpointResolverAdvertisement>
    <advertId>127.0.0.1-1082546529132-956257422</advertId>
    <peerId>127.0.0.1-1082546529051-721554874</peerId>
    <endpointAddress>
        <address>0:2223</address>
        <endpointType>unicast</endpointType>
        <transportProtocol>UDP</transportProtocol>
    </endpointAddress>
    <resolverPipeTypes>standard</resolverPipeTypes>
    <resolverPipeTypes>discovery</resolverPipeTypes>
    <resolverForPeerID>127.0.0.1-1082546529051-721554874</resolverForPeerID>
    <transportProtocol>UDP</transportProtocol>
</EndpointResolverAdvertisement>
PAINative: recv, Data Length 634
P2PSClient: Discovered a EndpointResolverAdvertisement
P2PSClient: Advert class is a p2ps.imp.endpoint.EndpointResolverAdvertisementImp
PAINative: setNs2Node
PAINative: dataReceivedAtNativeSocket
PAINative: setNs2Node
PAINative: notifying Listeners
PAINative: setDatagramDetails
PAINative: setDatagramDetails: Port - 5555 address is 1
PAINative: recv, Host Address 1
PAINative: recv, Port 5555
PAINative: recv, Data <?xml version="1.0" encoding="UTF-8"?>
<EndpointResolverAdvertisement>
    <advertId>127.0.0.1-1082546529540-214271659</advertId>
    <peerId>127.0.0.1-1082546529534-355209911</peerId>
    <endpointAddress>
        <address>1:2223</address>
        <endpointType>unicast</endpointType>
        <transportProtocol>UDP</transportProtocol>
    </endpointAddress>
    <resolverPipeTypes>standard</resolverPipeTypes>
    <resolverPipeTypes>discovery</resolverPipeTypes>
    <resolverForPeerID>127.0.0.1-1082546529534-355209911</resolverForPeerID>
    <transportProtocol>UDP</transportProtocol>
</EndpointResolverAdvertisement>
PAINative: recv, Data Length 634
PAINative: setNs2Node
PAINative: dataReceivedAtNativeSocket
PAINative: setNs2Node
PAINative: notifying Listeners
```

```
PAINative: setDatagramDetails
PAINative: setDatagramDetails: Port - 5555 address is 1
PAINative: recv, Host Address 1
PAINative: recv, Port 5555
PAINative: recv, Data <?xml version="1.0" encoding="UTF-8"?>
<EndpointResolverQuery>
    <query>true</query>
    <advertId>127.0.0.1-1082546529633-83243233</advertId>
    <peerId>127.0.0.1-1082546529534-355209911</peerId>
    <queryPeerID>127.0.0.1-1082546529534-355209911</queryPeerID>
    <replyEndpointAddress>
        <endpointAddress>
            <address>1:2223</address>
            <endpointType>unicast</endpointType>
            <transportProtocol>UDP</transportProtocol>
        </endpointAddress>
    </replyEndpointAddress>
</EndpointResolverQuery>
PAINative: recv, Data Length 552
PAINative: addPAISocketListener
PAINative: removeSocket
PAINative: removePAISocketListener
PAINative: setNs2Node
PAINative: dataReceivedAtNativeSocket
PAINative: setNs2Node
PAINative: notifying Listeners
PAINative: setDatagramDetails
PAINative: setDatagramDetails: Port - 5555 address is 0
PAINative: recv, Host Address 0
PAINative: recv, Port 5555
PAINative: recv, Data <?xml version="1.0" encoding="UTF-8"?>
<EndpointResolverQuery>
    <query>true</query>
    <advertId>127.0.0.1-1082546529308-453632564</advertId>
    <peerId>127.0.0.1-1082546529051-721554874</peerId>
    <queryPeerID>127.0.0.1-1082546529051-721554874</queryPeerID>
    <replyEndpointAddress>
        <endpointAddress>
            <address>0:2223</address>
            <endpointType>unicast</endpointType>
            <transportProtocol>UDP</transportProtocol>
        </endpointAddress>
    </replyEndpointAddress>
</EndpointResolverQuery>
PAINative: recv, Data Length 553
PAINative: addPAISocketListener
PAINative: removeSocket
PAINative: removePAISocketListener
P2PSClient: Discovered a EndpointResolverQuery
```

```
P2PSClient: Advert class is a p2ps.imp.endpoint.EndpointResolverQueryImp
PAINative: setNs2Node
PAINative: setNs2Node
P2PSServer: ------------------> Peer initialised
P2PSServer: ------------------> Server Started
PAINative: send, Host Address -2147483648
PAINative: send, Port 5555
PAINative: send, Data <?xml version="1.0" encoding="UTF-8"?>
<SRSS_Mobile_Node>
    <advertId>Mobile Node</advertId>
    <peerId>0</peerId>
    <STANDARD_UDP_PORT>3333</STANDARD_UDP_PORT>
</SRSS_Mobile_Node>
PAINative: send, Data Length 190
P2PSServer: ------------------> init finished... waiting
PAINative: setNs2Node
P2PSClient: ------------------> Started: Locating SRSS nodes
PAINative: send, Host Address -2147483648
PAINative: send, Port 5555
PAINative: send, Data <?xml version="1.0" encoding="UTF-8"?>
<SRSS_Mobile_Node>
    <query>true</query>
    <advertId>Mobile Node</advertId>
    <peerId></peerId>
</SRSS_Mobile_Node>
PAINative: send, Data Length 165
P2PSClient: Discovered a SRSS_Mobile_Node
P2PSClient: Advert class is a p2ps.impl.srss.SRSSQuery
PAINative: dataReceivedAtNativeSocket
PAINative: setNs2Node
PAINative: notifying Listeners
PAINative: setDatagramDetails
PAINative: setDatagramDetails: Port - 5555 address is 1
PAINative: recv, Host Address 1
PAINative: recv, Port 5555
PAINative: recv, Data <?xml version="1.0" encoding="UTF-8"?>
<SRSS_Mobile_Node>
    <query>true</query>
    <advertId>Mobile Node</advertId>
    <peerId></peerId>
</SRSS_Mobile_Node>
PAINative: recv, Data Length 165
PAINative: setNs2Node
PAINative: dataReceivedAtNativeSocket
PAINative: setNs2Node
PAINative: notifying Listeners
PAINative: setDatagramDetails
PAINative: setDatagramDetails: Port - 5555 address is 0
PAINative: recv, Host Address 0
```

```
PAINative: recv, Port 5555
PAINative: recv, Data <?xml version="1.0" encoding="UTF-8"?>
<SRSS_Mobile_Node>
    <advertId>Mobile Node</advertId>
    <peerId>0</peerId>
    <STANDARD_UDP_PORT>3333</STANDARD_UDP_PORT>
</SRSS_Mobile_Node>
PAINative: recv, Data Length 190
P2PSClient: Discovered a SRSS_Mobile_Node
P2PSClient: Advert class is a p2ps.impl.srss.SRSSAdvert
P2PSClient: ------------------> Advert discovered: 0 3333
PAINative: send, Host Address 0
PAINative: send, Port 3333
PAINative: send, Data Hello Proteus
PAINative: send, Data Length 13
PAINative: setNs2Node
PAINative: dataReceivedAtNativeSocket
PAINative: setNs2Node
PAINative: notifying Listeners
Receiving ---------------------------
PAINative: setDatagramDetails
PAINative: setDatagramDetails: Port - 5555 address is 1
PAINative: recv, Host Address 1
PAINative: recv, Port 3333
PAINative: recv, Data Hello Proteus
PAINative: recv, Data Length 13
P2PSServer: Received Hello Proteus from 1 port: 3333
PAINative: setNs2Node
[Ian-Taylors-Computer:examples/pai/javaAgent] scmijt%
```

Here, you can see the kind of interaction happening within NS2. P2PS is advertising and discovering several types of services dynamically behind-the-scenes in order to detect the capabilities of the other P2PS nodes it is communicating with. P2PS has to always quiz the resolvers of other P2PS nodes to find out which endpoint protocols they are using so that they can send the data in the appropriate manner.

The listing here illustrates these mechanisms and shows the *SRSSAdvert* being discovered on the P2PS NS2 network binding. It then sets the parameters (i.e. the address and port number) and sends the data to the server node. The server acknowledges receipt by printing out the message to the screen.

8.3 Conclusion

In this chapter, a brief overview of the P2PS integration was given. We then described, through the use of a discovery example, how P2PS could be accessed and used within Ns2 and how the various P2PS services are invoked. The example here, showed a simple decentralised discovery of a server from a simple client application using the P2PS discovery capabilities. Once the server had been discovered, the client used PAI to send data directly to the server. Further, an example screen dump of a NS run was provided in order to show the kinds of interactions taking place within this NS2 P2PS implementation.

Bibliography

[1] The Ns2 Simulator, see *http://www.isi.edu/nsnam/ns/*

[2] The Java Home Page, see *http://java.sun.com/*

[3] The Java Tutorial: A practical guide for programmers, *http://java.sun.com/docs/books/tutorial/*

[4] Jxta, see *http://www.Jxta.org/*

[5] The Gridlab Project, see *http://www.gridlab.org/*

[6] The GridOneD Project, see *http://www.gridoned.org/*

[7] Langley, A. The Trouble with JXTA, see *http://www.openp2p.com/pub/a/p2p/2001/05/02/jxta_trouble.html*

[8] P2PS, part of the the Triana Project, see*http://www.trianacode.org/*

[9] Part of the Protean project see*TODO*

[10] The Jini web site: *http://www.jini.org/*

[11] Gamma E et al. Design Patterns: Elements of Reusable Object-Oriented Software, 1994, publisher Addison-Wesley, ISBN: 0201633612

[12] UDDI Technical White Paper, UDDI.org, September 6, 2000, see website *http://www.uddi.org*

[13] Web Services Invocation Framework (WSIF), see website *http://ws.apache.org/wsif/*

Index